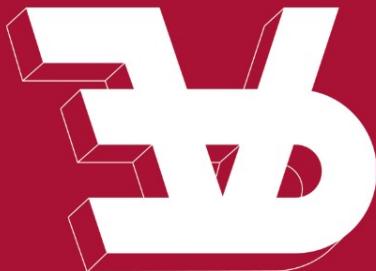


P. Madhusudan
Sanjit A. Seshia (Eds.)

LNCS 7358

Computer Aided Verification

24th International Conference, CAV 2012
Berkeley, CA, USA, July 2012
Proceedings



Commenced Publication in 1973

Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

P. Madhusudan
Sanjit A. Seshia (Eds.)

Computer Aided Verification

24th International Conference, CAV 2012
Berkeley, CA, USA, July 7-13, 2012
Proceedings

Volume Editors

P. Madhusudan

University of Illinois at Urbana-Champaign

Dept. of Computer Science

3226 Siebel Center, 201 N. Goodwin Avenue, Urbana, IL 61801-2302, USA

E-mail: madhu@illinois.edu

Sanjit A. Seshia

University of California, Berkeley

Dept. of Electrical Engineering and Computer Science

253 Cory Hall # 1770, Berkeley, CA 94720-1770, USA

E-mail: sseshia@eecs.berkeley.edu

ISSN 0302-9743

e-ISSN 1611-3349

ISBN 978-3-642-31423-0

e-ISBN 978-3-642-31424-7

DOI 10.1007/978-3-642-31424-7

Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012940389

CR Subject Classification (1998): D.2.4-5, I.2.2, F.3, F.1.1-2, F.4, C.3, B.3, D.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2012

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, 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.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 24th International Conference on Computer-Aided Verification (CAV) held in Berkeley, USA, July 7–3, 2012.

The Conference on Computer-Aided Verification (CAV) is dedicated to the advancement of the theory and practice of computer-aided formal methods for the analysis and synthesis of hardware, software, and other computational systems. Its scope ranges from theoretical results to concrete applications, with an emphasis on practical verification tools and the underlying algorithms and techniques.

The conference included two workshop days, a tutorial day, and three and a half days for the main program. We received 185 submissions (140 regular papers and 45 tool papers, a record number) and selected 38 regular and 20 tool papers. We appreciate the diligence of our Program Committee and our external reviewers, and thank them for their hard work; all papers received at least four reviews, and there was intense discussion on papers after the author response period.

This year CAV had four special tracks highlighted in the program: Computer Security, Embedded Systems, Hardware Verification, and SAT & SMT. We thank our Special Track Chairs for their effort in attracting papers in these areas and coordinating the review process for those papers.

The conference was preceded by seven affiliated workshops: The 5th International Workshop on Numerical Software Verification (NSV 2012); The First International Workshop on Memory Consistency Models (REORDER 2012); The 5th International Workshop on Exploiting Concurrency Efficiently and Correctly (EC2 2012); The Second International Workshop on Intermediate Verification Languages (BOOGIE 2012); The First Workshop on Logics for System Analysis (LfSA 2012); The First Workshop on Synthesis (SYNT 2012); The First Workshop on Applications of Formal Methods in Systems Biology (AFMSB 2012).

In addition to the presentations for the accepted papers, the conference also featured three invited talks and four invited tutorials.

– Invited talks:

- Wolfgang Thomas (RWTH, Aachen): “Synthesis and Some of Its Challenges”
- David Dill (Stanford University): “Model Checking Cell Biology”
- J. Alex Halderman (University of Michigan): “On Security of Voting Machines”

– Invited tutorials:

- Ras Bodik (University of California, Berkeley): “Synthesizing Programs with Constraint Solvers”
- Aaron Bradley (University of Colorado at Boulder): “IC3 and Beyond: Incremental, Inductive Verification”

- Chris Myers (University of Utah): “Formal Verification of Genetic Circuits”
- Michal Moskal (Microsoft) “From C to Infinity and Back: Unbounded Auto-active Verification with VCC”

We thank all our invited speakers!

We also thank the members of the CAV Steering Committee—Michael Gordon, Orna Grumberg, Bob Kurshan, and Ken McMillan—for their advice on various organizational matters. Shuvendu Lahiri, our Workshop Chair, smoothly handled the organization of the workshops. Miyoko Tsubamoto played an invaluable role in handling local arrangements. We thank Bryan Brady for his service as Publicity Chair and Edgar Pek for maintaining the website. Special thanks go to the Past Chairs, Ganesh Gopalakrishnan and Shaz Qadeer, for their advice and guidance throughout the process. We thank Alfred Hofmann and Anna Kramer of Springer for publishing the paper and USB proceedings for CAV 2012. We are grateful to Andrei Voronkov and his team for the use of the EasyChair system for tracking reviews and preparing the final camera-ready version. We gratefully acknowledge the donations provided by our corporate sponsors—Microsoft Research, IBM Research, Coverity, NEC Labs, and Intel. And last, but not the least, we thank the office staff of EECS Department at the University of California, Berkeley, and the Department of Computer Science at the University of Illinois at Urbana-Champaign, for providing critical administrative assistance in organizing the conference.

May 2012

P. Madhusudan
Sanjit A. Seshia

Organization

Program Committee

Rajeev Alur	University of Pennsylvania, USA
Roderick Bloem	Graz University of Technology, Austria
Supratik Chakraborty	IIT Bombay, India
Swarat Chaudhuri	Rice University, USA
Adam Chlipala	MIT, USA
Vincent Danos	University of Edinburgh, UK
Thomas Dillig	College of William and Mary, USA
Andy Gordon	Microsoft Research
Mike Gordon	University of Cambridge, UK
Orna Grumberg	Technion - Israel Institute of Technology, Israel
Aarti Gupta	NEC Labs America, USA
William Hung	Synopsys Inc.
Somesh Jha	University of Wisconsin, Madison, USA
Ranjit Jhala	University of California, San Diego, USA
Bengt Jonsson	Uppsala University, Sweden
Rajeev Joshi	NASA JPL
Daniel Kroening	Oxford University, UK
Andreas Kuehlmann	Coverity
Viktor Kuncak	EPFL, Switzerland
Shuvendu Lahiri	Microsoft Research
P. Madhusudan	University of Illinois, Urbana-Champaign, USA
Rupak Majumdar	MPI-SWS
Ken Mcmillan	Microsoft Research
David Molnar	Microsoft Research
Kedar Namjoshi	Bell Labs
Albert Oliveras	Technical University of Catalonia, Spain
Joel Ouaknine	Oxford University, UK
Gennaro Parlato	University of Southampton, UK
Nir Piterman	University of Leicester, UK
Andreas Podelski	University of Freiburg, Germany
Shaz Qadeer	Microsoft Research
Zvonimir Rakamaric	University of Utah, USA
Sriram Sankaranarayanan	University of Colorado, Boulder, USA
Sanjit A. Seshia	University of California, Berkeley, USA
Natasha Sharygina	University of Lugano, Switzerland
Stavros Tripakis	University of California, Berkeley, USA
Helmut Veith	Vienna University of Technology, Austria
Mahesh Viswanathan	University of Illinois, Urbana-Champaign, USA
Jin Yang	Intel Corporation
Karen Yorav	IBM Haifa Research Lab, Israel

Additional Reviewers

Abio, Ignasi
Adir, Allon
Akshay, S.
Alberti, Francesco
Atig, Mohamed Faouzi
Bennett, Huxley
Bing, Liu
Bingham, Jesse
Bogomolov, Sergiy
Boigelot, Bernard
Bruttomesso, Roberto
Bustan, Doron
Calin, Georgel
Chadha, Rohit
Chatterjee, Debapriya
Chen, Yu-Fang
Chockler, Hana
Cimatti, Alessandro
D'Solva, Vijay
Dang, Thao
Darulova, Eva
Davidson, Drew
Demyanova, Yulia
Donaldson, Alastair
Duggirala, Parasara Sridhar
Eisner, Cindy
Emmi, Michael
Enea, Constantin
Esmaeilsabzali, Shahram
Faouzi, Mohamed
Farzan, Azadeh
Fedyukovich, Grigory
Finkbeiner, Bernd
Fischer, Bernd
Florian, Mihai
Fredrikson, Matt
Frehse, Goran
Ganty, Pierre
Garg, Pranav
Gay, Simon
Geilen, Marc
German, Steven
Giannakopoulou, Dimitra
Goubault, Eric
Griesmayer, Andreas
Grinchtein, Olga
Groce, Alex
Guan, Nan
Gurfinkel, Arie
Habermehl, Peter
Hariharan, Ramesh
Harris, William
Harrison, John
Hofferek, Georg
Holcomb, Daniel
Holzer, Andreas
Holzmann, Gerard
Iosif, Radu
Ivrii, Alexander
Jacobs, Swen
Jha, Sumit Kumar
Jha, Susmit
Jhala, Ranjit
Jobstmann, Barbara
John, Annu
Keidar Barner, Sharon
Khalimov, Ayrat
Kini, Dileep
Kneuss, Etienne
Koenighofer, Robert
Konnov, Igor
Korchemny, Dmitry
Krstic, Sava
Kumar, Pratyush
La Torre, Salvatore
Lal, Akash
Legay, Axel
Leonardsson, Carl
Leroux, Jerome
Lewis, Matt
Li, Wenchao
Lodaya, Kamal
Logozzo, Francesco
Luchangco, Victor
Luchaup, Daniel
Matsliah, Arie

Miné, Antoine	Sheinvald, Sarai
Monniaux, David	Shoham, Sharon
Nadel, Alexander	Shurek, Gil
Narayanaswamy, Ganesh	Sighireanu, Mihaela
Nevo, Ziv	Sinha, Rohit
Nickovic, Dejan	Sinn, Moritz
Orni, Avigail	Sosnovich, Adi
Pandav, Sudhindra	Spielmann, Andrej
Parker, David	Srivastava, Saurabh
Parkinson, Matthew	Stenman, Jari
Persson, Magnus	Stigge, Martin
Pill, Ingo	Strichman, Ofer
Piskac, Ruzica	Suter, Philippe
Platzer, André	Talupur, Murali
Prabhu, Prathmesh	Tautschnig, Michael
Qian, Kairong	Terauchi, Tachio
Rajamani, Sriram	Tiwari, Ashish
Rajan, Ajitha	Tristan, Jean-Baptiste
Raskin, Jean-Francois	Tsitovich, Aliaksei
Ray, Sandip	Tuerk, Thomas
Rezine, Othmane	Vizel, Yakir
Riener, Heinz	Wahl, Thomas
Rodriguez-Carbonell, Enric	Wang, Bow-Yaw
Rollini, Simone Fulvio	Wang, Chao
Ruah, Sitvanit	Wasson, Zach
Rubio, Albert	Weissenbacher, Georg
Ruemmer, Philipp	Welp, Tobias
Rungta, Neha	Widder, Josef
Rybalchenko, Andrey	Wintersteiger, Christoph
Sadrzadeh, Mehrnoosh	Worrell, James
Samanta, Roopsha	Yi, Wang
Sangnier, Arnaud	Yu, Andy
Schewe, Sven	Zamfir, Cristian
Sery, Ondrej	Zuleger, Florian
Shashidhar, K.C.	Zuliani, Paolo

Table of Contents

Invited Talks

Synthesis and Some of Its Challenges	1
<i>Wolfgang Thomas</i>	
Model Checking Cell Biology	2
<i>David L. Dill</i>	

Invited Tutorials

Synthesizing Programs with Constraint Solvers	3
<i>Rastislav Bodík and Emina Torlak</i>	
IC3 and beyond: Incremental, Inductive Verification	4
<i>Aaron R. Bradley</i>	
Formal Verification of Genetic Circuits	5
<i>Chris J. Myers</i>	
From C to Infinity and Back: Unbounded Auto-active Verification with VCC	6
<i>Michał Moskal</i>	

Automata and Synthesis

Deterministic Automata for the (F,G)-Fragment of LTL	7
<i>Jan Křetínský and Javier Esparza</i>	
Efficient Controller Synthesis for Consumption Games with Multiple Resource Types	23
<i>Tomáš Brázdil, Krishnendu Chatterjee, Antonín Kučera, and Petr Novotný</i>	
ACTL \cap LTL Synthesis	39
<i>Rüdiger Ehlers</i>	

Inductive Inference and Termination

Learning Boolean Functions Incrementally	55
<i>Yu-Fang Chen and Bow-Yaw Wang</i>	
Interpolants as Classifiers	71
<i>Rahul Sharma, Aditya V. Nori, and Alex Aiken</i>	

Termination Analysis with Algorithmic Learning	88
<i>Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi</i>	
Automated Termination Proofs for Java Programs with Cyclic Data	105
<i>Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl</i>	
Proving Termination of Probabilistic Programs Using Patterns.....	123
<i>Javier Esparza, Andreas Gaiser, and Stefan Kiefer</i>	

Abstraction

The Gauge Domain: Scalable Analysis of Linear Inequality Invariants ...	139
<i>Arnaud J. Venet</i>	
Diagnosing Abstraction Failure for Separation Logic-Based Analyses ...	155
<i>Josh Berdine, Arlen Cox, Samin Ishtiaq, and Christoph M. Wintersteiger</i>	
A Method for Symbolic Computation of Abstract Operations	174
<i>Aditya Thakur and Thomas Reps</i>	
Leveraging Interpolant Strength in Model Checking	193
<i>Simone Fulvio Rollini, Ondrej Sery, and Natasha Sharygina</i>	

Concurrency and Software Verification

Detecting Fair Non-termination in Multithreaded Programs	210
<i>Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal</i>	
Lock Removal for Concurrent Trace Programs	227
<i>Vineet Kahlon and Chao Wang</i>	
How to Prove Algorithms Linearisable	243
<i>Gerhard Schellhorn, Heike Wehrheim, and John Derrick</i>	
Synchronisation- and Reversal-Bounded Analysis of Multithreaded Programs with Counters	260
<i>Matthew Hague and Anthony Widjaja Lin</i>	
Software Model Checking via IC3	277
<i>Alessandro Cimatti and Alberto Griggio</i>	

Biology and Probabilistic Systems

Delayed Continuous-Time Markov Chains for Genetic Regulatory Circuits	294
<i>Călin C. Guet, Ashutosh Gupta, Thomas A. Henzinger, Maria Mateescu, and Ali Sezgin</i>	

Assume-Guarantee Abstraction Refinement for Probabilistic Systems ... <i>Anvesh Komuravelli, Corina S. Păsăreanu, and Edmund M. Clarke</i>	310
---	-----

Cross-Entropy Optimisation of Importance Sampling Parameters for Statistical Model Checking <i>Cyrille Jegourel, Axel Legay, and Sean Sedwards</i>	327
--	-----

Embedded and Control Systems

Timed Relational Abstractions for Sampled Data Control Systems <i>Aditya Zutshi, Sriram Sankaranarayanan, and Ashish Tiwari</i>	343
--	-----

Approximately Bisimilar Symbolic Models for Digital Control Systems <i>Rupak Majumdar and Majid Zamani</i>	362
--	-----

Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System <i>Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdy, Tiziana Rizzo, Marco Roveri, Angela Sansevieri, and Andrei Tchaltsev</i>	378
--	-----

SAT/SMT Solving and SMT-based Verification

Minimum Satisfying Assignments for SMT <i>Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Alex Aiken</i>	394
--	-----

When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way <i>Cheng-Shen Han and Jie-Hong Roland Jiang</i>	410
---	-----

A Solver for Reachability Modulo Theories <i>Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri</i>	427
--	-----

Timed and Hybrid Systems

On Decidability of Prebisimulation for Timed Automata <i>Shibashis Guha, Chinmay Narayan, and S. Arun-Kumar</i>	444
--	-----

Exercises in Nonstandard Static Analysis of Hybrid Systems <i>Ichiro Hasuo and Kohei Suenaga</i>	462
---	-----

A Box-Based Distance between Regions for Guiding the Reachability Analysis of SpaceEx <i>Sergiy Bogomolov, Goran Frehse, Radu Grosu, Hamed Ladan, Andreas Podelski, and Martin Wehrle</i>	479
---	-----

Hardware Verification

An Axiomatic Memory Model for POWER Multiprocessors	495
<i>Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M.K. Martin, Peter Sewell, and Derek Williams</i>	
nuTAB-BackSpace: Rewriting to Normalize Non-determinism in Post-silicon Debug Traces	513
<i>Flavio M. De Paula, Alan J. Hu, and Amir Nahir</i>	
Incremental, Inductive CTL Model Checking	532
<i>Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi</i>	

Security

Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement	548
<i>Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas Reps, Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran</i>	
Automatic Quantification of Cache Side-Channels	564
<i>Boris Köpf, Laurent Mauborgne, and Martín Ochoa</i>	
Secure Programming via Visibly Pushdown Safety Games	581
<i>William R. Harris, Somesh Jha, and Thomas Reps</i>	

Verification and Synthesis

Alternate and Learn: Finding Witnesses without Looking All over	599
<i>Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan</i>	
A Complete Method for Symmetry Reduction in Safety Verification	616
<i>Duc-Hiep Chu and Joxan Jaffar</i>	
Synthesizing Number Transformations from Input-Output Examples....	634
<i>Rishabh Singh and Sumit Gulwani</i>	

Tool Demonstration Papers

Acacia+, a Tool for LTL Synthesis	652
<i>Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin</i>	
MGSyn: Automatic Synthesis for Industrial Automation	658
<i>Chih-Hong Cheng, Michael Geisinger, Harald Ruess, Christian Buckl, and Alois Knoll</i>	

OpenNWA: A Nested-Word Automaton Library	665
<i>Evan Driscoll, Aditya Thakur, and Thomas Reps</i>	
UFO: A Framework for Abstraction- and Interpolation-Based Software Verification	672
<i>Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik</i>	
SAFARI: SMT-Based Abstraction for Arrays with Interpolants	679
<i>Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina</i>	
BMA: Visual Tool for Modeling and Analyzing Biological Networks	686
<i>David Benque, Sam Bourton, Caitlin Cockerton, Byron Cook, Jasmin Fisher, Samin Ishtiaq, Nir Piterman, Alex Taylor, and Moshe Y. Vardi</i>	
APEX: An Analyzer for Open Probabilistic Programs	693
<i>Stefan Kiefer, Andrzej S. Murawski, Joël Ouaknine, Björn Wachter, and James Worrell</i>	
Recent Developments in FDR	699
<i>Philip Armstrong, Michael Goldsmith, Gavin Lowe, Joël Ouaknine, Hristina Palikareva, A.W. Roscoe, and James Worrell</i>	
A Model Checker for Hierarchical Probabilistic Real-Time Systems	705
<i>Songzheng Song, Jun Sun, Yang Liu, and Jin Song Dong</i>	
SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs	712
<i>Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo</i>	
Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems: Tool Paper	718
<i>Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi</i>	
HybridSAL Relational Abstracter	725
<i>Ashish Tiwari</i>	
EULER: A System for Numerical Optimization of Programs	732
<i>Swarat Chaudhuri and Armando Solar-Lezama</i>	
SPT: Storyboard Programming Tool	738
<i>Rishabh Singh and Armando Solar-Lezama</i>	
CSolve: Verifying C with Liquid Types	744
<i>Patrick Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala</i>	

PASSERT: A Tool for Debugging Parallel Programs	751
<i>Daniel Schwartz-Narbonne, Feng Liu, David August, and Sharad Malik</i>	
TRACER: A Symbolic Execution Tool for Verification	758
<i>Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa</i>	
Joogie: Infeasible Code Detection for Java	767
<i>Stephan Arlt and Martin Schäf</i>	
HECTOR: An Equivalence Checker for a Higher-Order Fragment of ML	774
<i>David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong</i>	
Resource Aware ML	781
<i>Jan Hoffmann, Klaus Aehlig, and Martin Hofmann</i>	
Author Index	787

Synthesis and Some of Its Challenges

Wolfgang Thomas

RWTH Aachen University, Lehrstuhl Informatik 7, 52056 Aachen, Germany
thomas@informatik.rwth-aachen.de

Keywords: Infinite games, winning strategies, uniformization problem, model-checking.

The advent of a methodology of automatic synthesis (of state-based systems) adds a number of interesting facets to the setting of model-checking. In this talk we focus on some conceptual aspects arising from the basic scenario of strategy synthesis in infinite-duration two-player games, as a natural extension of model-checking. The starting point is the simple observation that model-checking asks about the (non-) emptiness of sets while synthesis asks for a certain kind of uniformization of relations by functions. This raises a large number of questions on the classification of (word-) functions (which serve as strategies in games). We discuss basic results and recent progress, emphasizing two aspects: the definability of strategies and their "complexity" in various dimensions. These results are as yet preliminary, and we end by listing unresolved problems, for example on the logic-representation of strategies.

Model Checking Cell Biology

David L. Dill

Stanford University
dill@cs.stanford.edu

Abstract. Mathematical models of real biological systems have predominantly been deterministic or stochastic continuous models. However, there are reasons to believe that at least some processes can be modeled in a “digital” way. Once we do that, we enter the domain of concurrent and reactive systems, where model checking has been an important tool. Perhaps techniques from the verification community could lead to insights about the systems principles that allow biological systems using very low energy (and high noise) components to function dynamic environments.

I will explore some past and future research directions in this area, as well as some of the non-computational challenges that arise in this kind of research.

Synthesizing Programs with Constraint Solvers

Rastislav Bodík and Emina Torlak

University of California, Berkeley

Abstract. Classical synthesis derives programs from a specification. We show an alternative approach where programs are obtained through search in a space of candidate programs. Searching for a program that meets a specification frees us from having to develop a sufficiently complete set of derivation rules, a task that is more challenging than merely describing the syntactic shape of the desired program. To make the search for a program efficient, we exploit symbolic constraint solving, lifted to synthesis from the setting of program verification.

We start by describing the interface to the synthesizer, which the programmer uses to specify the space of candidate programs \mathcal{P} as well as the desired correctness condition ϕ . The space \mathcal{P} is defined by a program template whose missing expressions are described with a grammar. The correctness condition is a multi-modal specification, given as a combination of assertions, input / output pairs, and traces.

Next, we describe several algorithms for solving the synthesis problem $\exists P \forall x \phi(x, P(x))$. The key idea is to reduce the problem from 2QBF to SAT by sampling the space of inputs, which eliminates the universal quantification over x .

Finally, we show how to encode the resulting SAT problem in relational logic, and how this encoding can be used to solve a range of related problems that arise in synthesis, from verification to program state repair. We will conclude with open problems on constraint-based synthesis.

IC3 and beyond: Incremental, Inductive Verification

Aaron R. Bradley

ECEE Department, University of Colorado at Boulder
bradleya@colorado.edu

IC3, a SAT-based safety model checking algorithm introduced in 2010 [1] [2], is considered among the best safety model checkers. This tutorial discusses its essential ideas: the use of concrete states, called counterexamples to induction, to motivate lemma discovery; the incremental application of induction to generate the lemmas; and the use of stepwise assumptions to allow dynamic shifting between inductive lemma generation and propagation of lemmas as predicates.

Two perspectives on IC3 are offered: IC3 as proof finder, which highlights its ability to find mutually inductive lemmas, a crucial element of its robustness; and IC3 as bug finder, which shows that IC3's choices with respect to proof obligations result in a heuristically guided search. The latter perspective casts lemmas as refinements of estimates of states' proximities to initial states. These estimates guide the backward construction of potential counterexample traces.

IC3's context is then discussed: its evolution from earlier work and how it compares to other algorithms. Finally, the broader idea of incremental, inductive verification (IIV), of which IC3 is just one example, is explored. The IIV perspective has motivated new algorithms for analyzing ω -regular properties [4] and CTL properties [5].

A recent tutorial paper [3] provides a conceptual exposition of IC3, while an earlier tutorial paper [6] illustrates IC3's workings through detailed examples.

References

- [1] Bradley, A.R.: k-step relative inductive generalization. Technical report, CU Boulder (March 2010), <http://arxiv.org/abs/1003.3649>
- [2] 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)
- [3] Bradley, A.R.: Understanding IC3. In: SAT (June 2012)
- [4] Bradley, A.R., Somenzi, F., Hassan, Z., Zhang, Y.: An incremental approach to model checking progress properties. In: FMCAD (November 2011)
- [5] Hassan, Z., Bradley, A.R., Somenzi, F.: Incremental, Inductive CTL Model Checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, p. 4. Springer, Heidelberg (2012)
- [6] Somenzi, F., Bradley, A.R.: IC3: Where monolithic and incremental meet. In: FMCAD (November 2011)

Formal Verification of Genetic Circuits*

Chris J. Myers

University of Utah
myers@ece.utah.edu

Abstract. Researchers are beginning to be able to engineer synthetic genetic circuits for a range of applications in the environmental, medical, and energy domains [1]. Crucial to the success of these efforts is the development of methods and tools to verify the correctness of these designs. This verification though is complicated by the fact that genetic circuit components are inherently noisy making their behavior asynchronous, analog, and stochastic in nature [2]. Therefore, rather than definite results, researchers are often interested in the probability of the system reaching a given state within a certain amount of time. Usually, this involves simulating the system to produce some time series data and analyzing this data to discern the state probabilities. However, as the complexity of models of genetic circuits grow, it becomes more difficult for researchers to reason about the different states by looking only at time series simulation results of the models. To address this problem, techniques from the formal verification community, such as stochastic model checking, can be leveraged [3,4]. This tutorial will introduce the basic biology concepts needed to understand genetic circuits, as well as, the modeling and analysis techniques currently being employed. Finally, it will give insight into how formal verification techniques can be applied to genetic circuits.

References

1. Lucks, J., Arkin, A.: The hunt for the biological transistor. *IEEE Spectrum* 48(3), 38–43 (2011)
2. Elowitz, M.B., Levine, A.J., Siggia, E.D., Swain, P.S.: Stochastic gene expression in a single cell. *Science* 297(5584), 1183–1186 (2002)
3. Madsen, C., Myers, C., Patterson, T., Roehner, N., Stevens, J., Winstead, C.: Design and test of genetic circuits using iBioSim. *IEEE Design and Test of Computers* 29(3) (May/June 2012)
4. Madsen, C., Myers, C., Roehner, N., Winstead, C., Zhang, Z.: Utilizing stochastic model checking to analyze genetic circuits. In: *IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology* (2012)

* This work is supported by the National Science Foundation under Grants CCF-0916105 and CCF-0916042.

From C to Infinity and Back: Unbounded Auto-active Verification with VCC

Michał Moskal

Microsoft Research Redmond

michal.moskal@microsoft.com

Abstract. In this tutorial I'll show how to prove deep functional properties of tricky sequential and concurrent C programs using VCC. I'll get into induction, termination, algebraic data types, infinite maps, and lemmas, all unified as ghost data and C-like code manipulating it. Once these are provided, verification is automatic, but the development process of such annotations tends to be very interactive, thus "auto-active verification" using C as a proof language.

VCC [1] is an industrial-strength verification environment for low-level concurrent systems code written in C. VCC takes a program (annotated with function contracts, state assertions, and type invariants) and attempts to prove the correctness of these annotations. VCC's verification methodology [3] allows global two-state invariants that restrict update of shared state and enforces simple, semantic conditions sufficient for checking those global invariants modularly. VCC works by translating C, via the Boogie intermediate verification language, to verification conditions handled by the Z3 SMT solver.

The environment includes tools for monitoring proof attempts and constructing partial counterexample executions for failed proofs and has been used to verify functional correctness of tens of thousands of lines of Microsoft's Hyper-V virtualization platform and of SYSGO's embedded real-time operating system PikeOS.

VCC is available with sources for non-commercial use at <http://vcc.codeplex.com/>, and online at <http://rise4fun.com/Vcc>. A tutorial [2] is also provided.

References

1. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
2. Cohen, E., Hillebrand, M.A., Moskal, M., Schulte, W., Tobies, S.: Verifying C programs: A VCC tutorial. Working Draft, <http://vcc.codeplex.com/>
3. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local Verification of Global Invariants in Concurrent Programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 480–494. Springer, Heidelberg (2010)

Deterministic Automata for the (F,G) -Fragment of LTL

Jan Křetínský^{1,2,*} and Javier Esparza¹

¹ Fakultät für Informatik, Technische Universität München, Germany
`{jan.kretinsky,esparza}@in.tum.de`

² Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract. When dealing with linear temporal logic properties in the setting of e.g. games or probabilistic systems, one often needs to express them as deterministic omega-automata. In order to translate LTL to deterministic omega-automata, the traditional approach first translates the formula to a non-deterministic Büchi automaton. Then a determinization procedure such as of Safra is performed yielding a deterministic ω -automaton. We present a direct translation of the (F,G) -fragment of LTL into deterministic ω -automata with no determinization procedure involved. Since our approach is tailored to LTL, we often avoid the typically unnecessarily large blowup caused by general determinization algorithms. We investigate the complexity of this translation and provide experimental results and compare them to the traditional method.

1 Introduction

The ω -regular languages play a crucial role in formal verification of linear time properties, both from a theoretical and a practical point of view. For model-checking purposes one can comfortably represent them using nondeterministic Büchi automata (NBW), since one only needs to check emptiness of the intersection of two NBWs corresponding to the system and the negation of the property, and NBWs are closed under intersection. However, two increasingly important problems require to represent ω -regular languages by means of *deterministic* automata. The first one is synthesis of reactive modules for LTL specifications, which was theoretically solved by Pnueli and Rosner more than 20 years ago [PR88], but is recently receiving a lot of attention (see the references below). The second one is model checking Markov decision processes (see e.g. [BK08]), where impressive advances in algorithmic development and tool support are quickly extending the range of applications.

It is well known that NBWs are strictly more expressive than their deterministic counterpart, and so cannot be determinized. The standard theoretical solution to this problem is to translate NBW into deterministic Rabin automata (DRW) using Safra's construction [Saf88] or a recent improvement by Piterman

* The author is a holder of Brno PhD Talent Financial Aid and is supported by the Czech Science Foundation, grant No. P202/12/G061.

[Pit06]. However, it is commonly accepted that Safra’s construction is difficult to handle algorithmically due to its “messy state space” [Kup12]. Many possible strategies for solving this problem have been investigated. A first one is to avoid Safra’s construction altogether. A Safraless approach that reduces the synthesis problem to emptiness of nondeterministic Büchi tree automata has been proposed in [KV05, KPV06]. The approach has had considerable success, and has been implemented in [JB06]. Another strategy is to use heuristics to improve Safra’s construction, a path that has been followed in [KB06, KB07] and has produced the ltl2dstar tool [Kle]. Finally, a third strategy is to search for more efficient or simpler algorithms for subclasses of ω -regular languages. A natural choice is to investigate classes of LTL formulas. While LTL is not as expressive as NBW, the complexity of the translation of LTL to DRW still has $2^{2^{\Theta(n)}}$ complexity [KR10]. However, the structure of NBWs for LTL formulas can be exploited to construct a symbolic description of a deterministic parity automaton [MS08]. Fragments of LTL have also been studied. In [AT04], single exponential translations for some simple fragments are presented. Piterman et al. propose in [PPS06] a construction for reactivity(1) formulas that produces in cubic time a symbolic representation of the automaton. The construction has been implemented in the ANZU tool [JGWB07].

Despite this impressive body of work, the problem cannot yet be considered solved. This is particularly so for applications to probabilistic model checking. Since probabilistic model checkers need to deal with linear arithmetic, they profit much less from sophisticated symbolic representations like those used in [PPS06, MS08], or from the Safraless approach which requires to use tree automata. In fact, to the best of our knowledge no work has been done so far in this direction. The most successful approach so far is the one followed by the ltl2dstar tool, which explicitly constructs a reduced DRW. In particular, the ltl2dstar has been reimplemented in PRISM [KNP11], the leading probabilistic model checker.

However, the work carried in [KB06, KB07] has not considered the development of specific algorithms for fragments of LTL. This is the question we investigate in this paper: is it possible to improve on the results of ltl2dstar by restricting attention to a subset of LTL? We give an affirmative answer by providing a very simple construction for the (\mathbf{F}, \mathbf{G}) -fragment of LTL, i.e., the fragment generated by boolean operations and the temporal operators \mathbf{F} and \mathbf{G} . Our construction is still double exponential in the worst case, but is algorithmically very simple. We construct a deterministic Muller automaton for a formula φ of the fragment with a very simple state space: boolean combinations of formulas of the closure of φ . This makes the construction very suitable for applying reductions based on logical equivalences: whenever some logical rule shows that two states are logically equivalent, they can be merged. (This fact is also crucial for the success of the constructions from LTL to NBW.) Since the number of Muller accepting sets can be very large, we also show that the Muller condition of our automata admits a compact representation as a generalized Rabin acceptance condition. We also show how to efficiently transform this automaton to a standard Rabin automaton. Finally, we report on an implementation of the

construction, and present a comparison with ltl2dstar. We show that our construction leads to substantially smaller automata for formulas expressing typical fairness conditions, which play a very important rôle in probabilistic model checking. For instance, while ltl2dstar produces an automaton with over one million states for the formula $\bigwedge_{i=1}^3 (\mathbf{G}Fa_i \rightarrow \mathbf{G}Fb_i)$, our construction delivers an automaton with 1560 states.

2 Linear Temporal Logic

This section recalls the notion of linear temporal logic (LTL) [Pnu77].

Definition 1 (LTL Syntax). *The formulae of the (\mathbf{F}, \mathbf{G}) -fragment of linear temporal logic are given by the following syntax:*

$$\varphi ::= a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi$$

where a ranges over a finite fixed set Ap of atomic propositions.

We use the standard abbreviations $\mathbf{tt} := a \vee \neg a$, $\mathbf{ff} := a \wedge \neg a$. We only have negations of atomic propositions, as negations can be pushed inside due to the equivalence of $\mathbf{F}\varphi$ and $\neg\mathbf{G}\neg\varphi$.

Definition 2 (LTL Semantics). *Let $w \in (2^{Ap})^\omega$ be a word. The i th letter of w is denoted $w[i]$, i.e. $w = w[0]w[1]\dots$. Further, we define the i th suffix of w as $w_i = w[i]w[i+1]\dots$. The semantics of a formula on w is then defined inductively as follows:*

$$\begin{array}{ll} w \models a & \iff a \in w[0] \\ w \models \neg a & \iff a \notin w[0] \\ w \models \varphi \wedge \psi & \iff w \models \varphi \text{ and } w \models \psi \\ w \models \varphi \vee \psi & \iff w \models \varphi \text{ or } w \models \psi \\ w \models \mathbf{F}\varphi & \iff \exists k \in \mathbb{N} : w_k \models \varphi \\ w \models \mathbf{G}\varphi & \iff \forall k \in \mathbb{N} : w_k \models \varphi \end{array}$$

We define a symbolic one-step unfolding \mathfrak{U} of a formula inductively by the following rules, where the symbol \mathbf{X} intuitively corresponds to the meaning of the standard next operator.

$$\begin{aligned} \mathfrak{U}(a) &= a \\ \mathfrak{U}(\neg a) &= \neg a \\ \mathfrak{U}(\varphi \wedge \psi) &= \mathfrak{U}(\varphi) \wedge \mathfrak{U}(\psi) \\ \mathfrak{U}(\varphi \vee \psi) &= \mathfrak{U}(\varphi) \vee \mathfrak{U}(\psi) \\ \mathfrak{U}(\mathbf{F}\varphi) &= \mathfrak{U}(\varphi) \vee \mathbf{X}\mathfrak{U}(\varphi) \\ \mathfrak{U}(\mathbf{G}\varphi) &= \mathfrak{U}(\varphi) \wedge \mathbf{X}\mathfrak{U}(\varphi) \end{aligned}$$

Example 3. Consider $\varphi = \mathbf{F}a \wedge \mathbf{G}Fb$. Then $\mathfrak{U}(\varphi) = (a \vee \mathbf{X}\mathfrak{U}(a)) \wedge (b \vee \mathbf{X}\mathfrak{U}(Fb)) \wedge \mathbf{X}\mathfrak{U}(\mathbf{G}Fb)$.

3 Deterministic Automaton for the (F,G) -Fragment

Let φ be an arbitrary but fixed formula. In the following, we construct a deterministic finite ω -automaton that recognizes the words satisfying φ . The definition of the acceptance condition and its variants follow in the subsequent sections. We start with a construction of the state space. The idea is that a state corresponds to a formula that needs to be satisfied when coming into this state. After evaluating the formulae on the propositions currently read, the next state will be given by what remains in the one-step unfold of the formula. E.g. for Example 3 and reading a , the successor state needs to satisfy $\mathbf{F}b \wedge \mathbf{G}\mathbf{F}b$.

In the classical syntactic model constructions, the states are usually given by sets of subformulae of φ . This corresponds to the conjunction of these subformulae. The main difference in our approach is the use of both conjunctions and also disjunctions that allow us to dispose of non-determinism in the corresponding transition function. In order to formalize this, we need some notation.

Let \mathbb{F} and \mathbb{G} denote the set of all subformulae of φ of the form $\mathbf{F}\psi$ and $\mathbf{G}\psi$, respectively. Further, all temporal subformulae are denoted by a shorthand $\mathbb{T} := \mathbb{F} \cup \mathbb{G}$. Finally, for a set of formulae Ψ , we denote $\mathbf{X}\Psi := \{\mathbf{X}\psi \mid \psi \in \Psi\}$.

We denote the *closure* of φ by $\mathbb{C}(\varphi) := Ap \cup \{\neg a \mid a \in Ap\} \cup \mathbf{X}\mathbb{T}$. Then $\mathfrak{U}(\varphi)$ is a positive Boolean combination over $\mathbb{C}(\varphi)$. By $\text{states}(\varphi)$ we denote the set $2^{2^{\mathbb{C}(\varphi)}}$. Each element of $\text{states}(\varphi)$ is a positive Boolean function over $\mathbb{C}(\varphi)$ and we often use a positive Boolean formula as its representative. For instance, the definition of \mathfrak{U} is clearly independent of the choice of representative, hence we abuse the notation and apply \mathfrak{U} to elements of $\text{states}(\varphi)$. Note that $|\text{states}(\varphi)| \in \mathcal{O}(2^{2^{|\varphi|}})$ where $|\varphi|$ denotes the length of φ .

Our state space has two components. Beside the logical component, we also keep track of one-step history of the word read. We usually use letters ψ, χ when speaking about the former component and α, β for the latter one.

Definition 4. Given a formula φ , we define $\mathcal{A}(\varphi) = (Q, i, \delta)$ to be a deterministic finite automaton over $\Sigma = 2^{Ap}$ given by

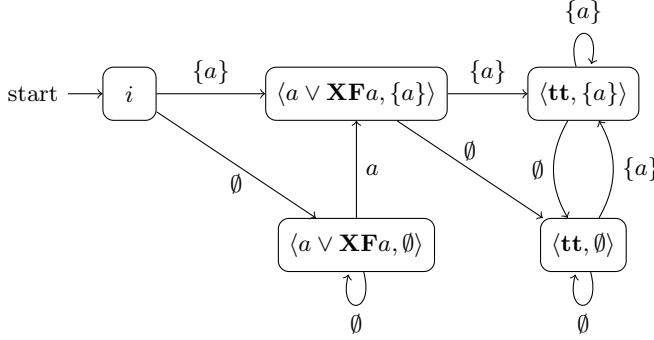
- the set of states $Q = \{i\} \cup (\text{states}(\varphi) \times 2^{Ap})$
- the initial state i ;
- the transition function

$$\delta = \{(i, \alpha, \langle \mathfrak{U}(\varphi), \alpha \rangle) \mid \alpha \in \Sigma\} \cup \{(\langle \psi, \alpha \rangle, \beta, \langle \text{succ}(\psi, \alpha), \beta \rangle) \mid \langle \psi, \alpha \rangle \in Q, \beta \in \Sigma\}$$

where $\text{succ}(\psi, \alpha) = \mathfrak{U}(\text{next}(\psi[\alpha \mapsto \mathbf{tt}, Ap \setminus \alpha \mapsto \mathbf{ff}]))$ where $\text{next}(\psi')$ removes \mathbf{X} 's from ψ' and $\psi[T \mapsto \mathbf{tt}, F \mapsto \mathbf{ff}]$ denotes the equivalence class of formulae where in ψ we substitute \mathbf{tt} for all elements of T and \mathbf{ff} for all elements of F .

Intuitively, a state $\langle \psi, \alpha \rangle$ of Q corresponds to the situation where ψ needs to be satisfied and α is being read.

Example 5. The automaton for \mathbf{Fa} with $Ap = \{a\}$ is depicted in the following figure. The automaton is obviously unnecessarily large, one can expect to merge e.g. the two states bearing the requirement \mathbf{tt} as the proposition a is irrelevant for satisfaction of \mathbf{tt} that does not even contain it. For the sake of simplicity, we leave all possible combinations here and comment on this in Section 8.



The reader might be surprised or even annoyed by the fact that the logical structure of the state space is not sufficient to keep enough information to decide whether a run ρ is accepting. In order to ensure this, we remember one-step history in the state. Why is that? Consider $\varphi = \mathbf{GF}(a \wedge \mathbf{F}b)$. Its unfold is then

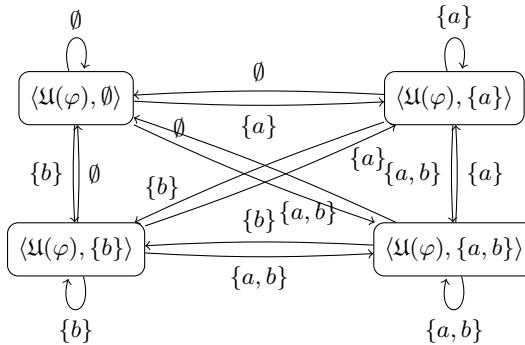
$$\mathbf{XGF}(a \wedge \mathbf{F}b) \wedge (\mathbf{XF}(a \wedge \mathbf{F}b) \vee (a \wedge (b \vee \mathbf{XF}b))) \quad (*)$$

Then moving under $\{a\}$ results into the requirement $\mathbf{GF}(a \wedge \mathbf{F}b) \wedge (\mathbf{F}(a \wedge \mathbf{F}b) \vee \mathbf{F}(b))$ for the next step where the alternative of pure $\mathbf{F}b$ signals progress made by not having to wait for an a . Nevertheless, the unfold of this formula is propositionally equivalent to (*). This is indeed correct as the two formulae are temporally equivalent (i.e. in LTL semantics). Thus, the information about the read a is not kept in the state and the information about this partial progress is lost! And now the next step under both $\{b\}$ and \emptyset again lead to the same requirement $\mathbf{GF}(a \wedge \mathbf{F}b) \wedge \mathbf{F}(a \wedge \mathbf{F}b)$. Therefore, there is no information that if b is read, then it can be matched with the previous a and we already have one satisfaction of (infinitely many required satisfactions of) $\mathbf{F}(a \wedge \mathbf{F}b)$ compared to reading \emptyset . Hence, the runs on $(\{a\}\{b\})^\omega$ and $(\{a\}\emptyset)^\omega$ are the same while the former should be accepting and the latter rejecting. However, this can be fixed by remembering the one-step history and using the acceptance condition defined in the following section.

4 Muller Acceptance Condition

In this section, we introduce a Muller acceptance condition. In general, the number of sets in a Muller condition can be exponentially larger than the size of the automaton. Therefore, we investigate the particular structure of the condition. In the next section, we provide a much more compact whilst still useful description of the condition. Before giving the formal definition, let us show an example.

Example 6. Let $\varphi = \mathbf{F}(\mathbf{G}a \vee \mathbf{G}b)$. The corresponding automaton is depicted below, for clarity, we omit the initial state. Observe that the formula stays the same and the only part that changes is the letter currently read that we remember in the state. The reason why is that φ can neither fail in finite time (there is always time to fulfill it), nor can be partially satisfied (no progress counts in this formula, only the infinite suffix). However, at some finite time the argument of \mathbf{F} needs to be satisfied. Although we cannot know when and whether due to $\mathbf{G}a$ or $\mathbf{G}b$, we know it is due to one of these (or both) happening. Thus we may shift the non-determinism to the acceptance condition, which says here: accept if the states where a holds are ultimately never left, or the same happens for b . The commitment to e.g. ultimately satisfying $\mathbf{G}a$ can then be proved by checking that all infinitely often visited states read a .



We now formalize this idea. Let φ be a formula and $\mathcal{A}(\varphi) = (Q, i, \delta)$ its corresponding automaton. Consider a formula χ as a Boolean function over elements of $\mathbb{C}(\varphi)$. For sets $T, F \subseteq \mathbb{C}(\varphi)$, let $\chi[T \mapsto \mathbf{tt}, F \mapsto \mathbf{ff}]$ denote the formula where \mathbf{tt} is substituted for elements of T , and \mathbf{ff} for F . As elements of $\mathbb{C}(\varphi)$ are considered to be atomic expressions here, the substitution is only done on the propositional level and does not go through the modality, e.g. $(a \vee \mathbf{X}G a)[a \rightarrow \mathbf{ff}] = \mathbf{ff} \vee \mathbf{X}G a$, which is equivalent to $\mathbf{X}G a$ in the propositional semantics.

Further, for a formula χ and $\alpha \in \Sigma$ and $I \subseteq \mathbb{T}$, we put $I \models_\alpha \chi$ to denote that

$$\chi[\alpha \cup I \mapsto \mathbf{tt}, Ap \setminus \alpha \mapsto \mathbf{ff}]$$

is equivalent to \mathbf{tt} in the propositional semantics. We use this notation to describe that we rely on a commitment to satisfy all formulae of I .

Definition 7 (Muller acceptance). A set $M \subseteq Q$ is **Muller accepting** for a set $I \subseteq \mathbb{T}$ if the following is satisfied:

1. for each $(\chi, \alpha) \in M$, we have $\mathbf{XI} \models_\alpha \chi$,
2. for each $\mathbf{F}\psi \in I$ there is $(\chi, \alpha) \in M$ with $I \models_\alpha \psi$,
3. for each $\mathbf{G}\psi \in I$ and for each $(\chi, \alpha) \in M$ we have $I \models_\alpha \psi$.

A set $F \subseteq Q$ is **Muller accepting** (for φ) if it is Muller accepting for some $I \subseteq \mathbb{T}$.

The first condition ensures that the commitment to formulae in I being ultimately satisfied infinitely often is enough to satisfy the requirements. The second one guarantees that each \mathbf{F} -formula is unfolded only finitely often and then satisfied, while the third one guarantees that \mathbf{G} -formulae indeed ultimately hold. Note that it may be impossible to see the satisfaction of a formula directly and one must rely on further promises, formulae of smaller size. In the end, promising the atomic proposition is not necessary and is proven directly from the second component of the state space.

4.1 Correctness

Given a formula φ , we have defined a Muller automaton $\mathcal{A}(\varphi)$ and we let the acceptance condition $\mathcal{M}(\varphi) = \{M_1, \dots, M_k\}$ be given by all the Muller accepting sets M_i for φ . Every word $w : \mathbb{N} \rightarrow 2^{Ap}$ induces a run $\rho = \mathcal{A}(\varphi)(w) : \mathbb{N} \rightarrow Q$ starting in i and following δ . The run is thus accepting and the word is accepted if the set of states visited infinitely often $\text{Inf}(\rho)$ is Muller accepting for φ . Vice versa, a run $\rho = i(\chi_1, \alpha_1)(\chi_2, \alpha_2) \dots$ induces a word $Ap(\rho) = \alpha_1 \alpha_2 \dots$. We now prove that this acceptance condition is sound and complete.

Theorem 8. *Let φ be a formula and w a word. Then w is accepted by the deterministic automaton $\mathcal{A}(\varphi)$ with the Muller condition $\mathcal{M}(\varphi)$ if and only if $w \models \varphi$.*

We start by proving that the first component of the state space takes care of all progress or failure in finite time.

Proposition 9 (Local (finitary) correctness). *Let w be a word and $\mathcal{A}(\varphi)(w) = i(\chi_0, \alpha_0)(\chi_1, \alpha_1) \dots$ the corresponding run. Then for all $n \in \mathbb{N}$, we have $w \models \varphi$ if and only if $w_n \models \chi_n$.*

Proof (Sketch). The one-step unfold produces a temporally equivalent (w.r.t. LTL satisfaction) formula. The unfold is a Boolean function over atomic propositions and elements of \mathbf{XT} . Therefore, this unfold is satisfied if and only if the next state satisfies $\text{next}(\psi)$ where ψ is the result of partial application of the Boolean function to the currently read letter of the word. We conclude by induction. \square

Further, each occurrence of satisfaction of \mathbf{F} must happen in finite time. As a consequence, a run with $\chi_i \not\models \mathbf{ff}$ is rejecting if and only if satisfaction of some $\mathbf{F}\psi$ is always postponed.

Proposition 10 (Completeness). *If $w \models \varphi$ then $\text{Inf}(\mathcal{A}(\varphi)(w))$ is a Muller accepting set.*

Proof. Let us show that $M := \text{Inf}(\mathcal{A}(\varphi)(w))$ is Muller accepting for

$$I := \{\psi \in \mathbb{F} \mid w \models \mathbf{G}\psi\} \cup \{\psi \in \mathbb{G} \mid w \models \mathbf{F}\psi\}$$

As a technical device we use the following. For every finite Boolean combination ψ of elements of the closure \mathbb{C} , there are only finitely many options to satisfy

it, each corresponding to a subset of \mathbb{C} . Therefore, if $w_i \models \psi$ for infinitely many $i \in \mathbb{N}$ then at least one of the options has to recur. More precisely, for some subset $\alpha \subseteq Ap$ there are infinitely many $i \in \mathbb{N}$ with $w_i \models \psi \cup \alpha \cup \{\neg a \mid a \in Ap \setminus \alpha\}$. For each such α we pick one subset $I_{\chi, \alpha} \subseteq \mathbb{T}$ such that for infinitely many i , after reading $w^i = w[0] \cdots w[i]$ we are in state (χ, α) and $w_i \models \psi \cup \mathbf{X}I_{\chi, \alpha}$, and $I_{\chi, \alpha} \models_{\alpha} \psi$. We say that we have a *recurring set* $I_{\chi, \alpha}$ *modelling* ψ (for a state (χ, α)). Obviously, the recurring sets for all states are included in I , i.e. $I_{\chi, \alpha} \subseteq I$ for every $(\chi, \alpha) \in Q$.

Let us now proceed with proving the three conditions of Definition 7 for M and I .

Condition 1. Let $(\chi, \alpha) \in M$. Since $w \models \varphi$, by Proposition 9 $w_i \models \chi$ whenever we enter (χ, α) after reading w^i , which happens for infinitely many $i \in \mathbb{N}$. Hence we have a recurring set $I_{\chi, \alpha}$ modelling χ . Since $I_{\chi, \alpha} \models_{\alpha} \chi$, we get also $I \models_{\alpha} \chi$ by $I_{\chi, \alpha} \subseteq I$.

Condition 2. Let $\mathbf{F}\psi \in I$, then $w \models \mathbf{GF}\psi$. Since there are finitely many states, there is $(\chi, \alpha) \in M$ for which after infinitely many entrances by w^i it holds $w_i \models \psi$ by Proposition 9, hence we have a recurring set $I_{\chi, \alpha}$ modelling ψ and conclude as above.

Condition 3. Let $\mathbf{G}\psi \in I$, then $w \models \mathbf{FG}\psi$. Hence for every $(\chi, \alpha) \in M$ infinitely many w^i leading to (χ, α) satisfy $w_i \models \psi$ by Proposition 9, hence we have a recurring set $I_{\chi, \alpha}$ modelling ψ and conclude as above. \square

Before proving the opposite direction of the theorem, we provide a property of Muller accepting sets opposite to the previous proposition.

Lemma 11. *Let ρ be a run. If $\text{Inf}(\rho)$ is Muller accepting for I then $Ap(\rho) \models \mathbf{G}\psi$ for each $\psi \in I \cap \mathbb{F}$ and $Ap(\rho) \models \mathbf{F}\psi$ for each $\psi \in I \cap \mathbb{G}$.*

Proof. Denote $w = Ap(\rho)$. Let us first assume $\psi \in I \cap \mathbb{F}$ and $w_j \not\models \psi$ for all $j \geq i \in \mathbb{N}$. Since $\psi \in I \cap \mathbb{F}$, for infinitely many j , ρ passes through some $(\chi, \alpha) \in \text{Inf}(\rho)$ for which $I \models_{\alpha} \psi$. Hence, there is $\psi_1 \in I$ which is a subformula of ψ such that for infinitely many i , $w_i \not\models \psi_1$. If $\psi_1 \in \mathbb{F}$, we proceed as above; similarly for $\psi_1 \in \mathbb{G}$. Since we always get a smaller subformula, at some point we obtain either $\psi_n = \mathbf{F}\beta$ or $\psi_n = \mathbf{G}\beta$ with β a Boolean combination over Ap and we get a contradiction with the second or the third point of Definition 7 respectively. \square

In other words, if we have a Muller accepting set for I then all elements of I hold true in w_i for almost all i .

Proposition 12 (Soundness). *If $\text{Inf}(\mathcal{A}(\varphi)(w))$ is a Muller accepting set then $w \models \varphi$.*

Proof. Let $M := \text{Inf}(\mathcal{A}(\varphi)(w))$ be a Muller accepting set for some I . There is $i \in \mathbb{N}$ such that after reading w^i we come to (χ, α) and stay in $\text{Inf}(\mathcal{A}(\varphi)(w))$ from now on and, moreover, $w_i \models \psi$ for all $\psi \in I$ by Lemma 11. For a contradiction, let $w \not\models \varphi$. By Proposition 9 we thus get $w_i \not\models \chi$. By the first condition of Definition 7, we get $I \models_{\alpha} \chi$. Therefore, there is $\psi \in I$ such that $w_i \not\models \psi$, a contradiction. \square

5 Generalized Rabin Condition

In this section, we investigate the structure of the previously defined Muller condition and propose a new type of acceptance condition that compactly, yet reasonably explicitly captures the accepting sets.

Let us first consider a fixed $I \subseteq \mathbb{T}$ and examine all Muller accepting sets for I . The first condition of Definition 7 requires not to leave the set of states $\{(\chi, \alpha) \mid I \models_\alpha \chi\}$. Similarly, the third condition is a conjunction of $|I \cap \mathbb{G}|$ conditions not to leave sets $\{(\chi, \alpha) \mid I \models_\alpha \psi\}$ for each $\mathbf{G}\psi \in I$. Both conditions thus together require that certain set (complement of the intersection of the above sets) is visited only finitely often. On the other hand, the second condition requires to visit certain sets infinitely often. Indeed, for each $\mathbf{F}\psi$ the set $\{(\chi, \alpha) \mid I \models_\alpha \psi\}$ must be visited infinitely often.

Furthermore, a set is accepting if the conditions above hold for *some* set I . Hence, the acceptance condition can now be expressed as a positive Boolean combination over Rabin pairs in a similar way as the standard Rabin condition is a disjunction of Rabin pairs.

Example 13. Let us consider the (strong) fairness constraint $\varphi = \mathbf{FG}a \vee \mathbf{GF}b$. Since each atomic proposition has both \mathbf{F} and \mathbf{G} as ancestors in the syntactic tree, it is easy to see that there is only one reachable element of states(φ) and the state space of \mathcal{A} is $\{i\} \cup 2^{\{a,b\}}$, i.e. of size $1 + 2^2 = 5$. Furthermore, the syntactic tree of $\mathfrak{U}(\varphi) = \mathbf{XFG}a \vee (\mathbf{XG}a \wedge a) \vee (\mathbf{XGF}b \wedge (\mathbf{XF}b \vee b))$ immediately determines possible sets I . These either contain $\mathbf{G}a$ (possibly with also $\mathbf{FG}a$ or some other elements) or $\mathbf{GF}b, \mathbf{F}b$. The first option generates the requirement to visit states with $\neg a$ only finitely often, the second one to visit b infinitely often. Thus the condition can be written as

$$(\{q \mid q \models \neg a\}, Q) \vee (\emptyset, \{q \mid q \models b\})$$

and is in fact a Rabin acceptance condition.

We formalize this new type of acceptance condition as follows.

Definition 14 (Generalized Rabin Automaton). A generalized Rabin automaton is a (deterministic) ω -automaton $\mathcal{A} = (Q, i, \delta)$ over some alphabet Σ , where Q is a set of states, i is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, together with a generalized Rabin condition $\mathcal{GR} \in \mathcal{B}^+(2^Q \times 2^Q)$. A run ρ of \mathcal{A} is accepting if $\text{Inf}(\rho) \models \mathcal{GR}$, which is defined inductively as follows:

$$\begin{aligned} \text{Inf}(\rho) \models \varphi \wedge \psi &\iff \text{Inf}(\rho) \models \varphi \text{ and } \text{Inf}(\rho) \models \psi \\ \text{Inf}(\rho) \models \varphi \vee \psi &\iff \text{Inf}(\rho) \models \varphi \text{ or } \text{Inf}(\rho) \models \psi \\ \text{Inf}(\rho) \models (F, I) &\iff F \cap \text{Inf}(\rho) = \emptyset \text{ and } I \cap \text{Inf}(\rho) \neq \emptyset \end{aligned}$$

The generalized Rabin condition corresponding to the previously defined Muller condition \mathcal{M} can now be formalized as follows.

Definition 15 (Generalized Rabin Acceptance). Let φ be a formula. The generalized Rabin condition $\mathcal{GR}(\varphi)$ is

$$\bigvee_{I \subseteq \mathbb{T}} \left(\left(\{(\chi, \alpha) \mid I \not\models_\alpha \chi \wedge \bigwedge_{\mathbf{G}\psi \in I} \psi\}, Q \right) \wedge \bigwedge_{\mathbf{F}\omega \in I} \left(\emptyset, \{(\chi, \alpha) \mid I \models_\alpha \omega\} \right) \right)$$

By the argumentation above, we get the equivalence of the Muller and the generalized Rabin conditions for φ and thus the following.

Proposition 16. Let φ be a formula and w a word. Then w is accepted by the deterministic automaton $\mathcal{A}(\varphi)$ with the generalized Rabin condition $\mathcal{GR}(\varphi)$ if and only if $w \models \varphi$.

Example 17. Let us consider a conjunction of two (strong) fairness constraints $\varphi = (\mathbf{FG}a \vee \mathbf{GF}b) \wedge (\mathbf{FG}c \vee \mathbf{GF}d)$. Since each atomic proposition is wrapped in either **FG** or **GF**, there is again only one relevant element of states(φ) and the state space of \mathcal{A} is $\{i\} \cup 2^{\{a,b,c,d\}}$, i.e. of size $1 + 2^4 = 17$. From the previous example, we already know the disjunctions correspond to $(\neg a, Q) \vee (\emptyset, b)$ and $(\neg c, Q) \vee (\emptyset, d)$. Thus for the whole conjunction, we get a generalized Rabin condition

$$((\neg a, Q) \vee (\emptyset, b)) \wedge ((\neg c, Q) \vee (\emptyset, d))$$

6 Rabin Condition

In this section, we briefly describe how to obtain a Rabin automaton from $\mathcal{A}(\varphi)$ and the generalized Rabin condition $\mathcal{GR}(\varphi)$ of Definition 15. For a fixed I , the whole conjunction of Definition 15 corresponds to the intersection of automata with different Rabin conditions. In order to obtain the intersection, one has first to construct the product of the automata, which in this case is still the original automaton with the state space Q , as they are all the same. Further, satisfying

$$(G, Q) \wedge \bigwedge_{f \in \mathcal{F} := I \cap \mathbb{F}} (\emptyset, F_f)$$

amounts to visiting G only finitely often and each F_f infinitely often. To check the latter (for a non-empty conjunction), it is sufficient to multiply the state space by \mathcal{F} with the standard trick that we leave the f th copy once we visit F_f and immediately go to the next copy. The resulting Rabin pair is thus

$$(G \times \mathcal{F}, F_{\bar{f}} \times \{\bar{f}\})$$

for an arbitrary fixed $\bar{f} \in \mathcal{F}$.

As for the disjunction, Rabin condition is closed under it as it simply takes the union of the pairs when the two automata have the same state space. In our case, one can multiply the state space of each disjunct corresponding to I by all

$J \cap \mathbb{F}$ for each $J \in 2^{\mathbb{T}} \setminus \{I\}$ to get the same state space for all of them. We thus get a bound for the state space

$$\prod_{I \subseteq \mathbb{T}} |I \cap \mathbb{F}| \cdot |Q|$$

Example 18. The construction of Definition 15 for the two fairness constraints Example 17 yields

$$(\neg a \vee \neg c, Q) \vee (\neg a, d) \vee (\neg c, b) \vee ((\emptyset, b) \wedge (\emptyset, d))$$

where we omitted all pairs (F, I) for which we already have a pair (F', I') with $F \subseteq F'$ and $I \supseteq I'$. One can eliminate the conjunction as described above at the cost of multiplying the state space by two. The corresponding Rabin automaton thus has $2 \cdot 1 \cdot |\{i\} \cup 2^{Ap}| = 34$ states. (Of course, for instance the initial state need not be duplicated, but for the sake of simplicity of the construction we avoid any optimizations.)

For a conjunction of three conditions, $\varphi = (\mathbf{F}G a \vee \mathbf{G}F b) \wedge (\mathbf{F}G c \vee \mathbf{G}F d) \wedge (\mathbf{F}G e \vee \mathbf{G}F f)$, the right components of the Rabin pairs correspond to $\mathbf{tt}, b, d, f, b \wedge d, b \wedge f, d \wedge f, b \wedge d \wedge f$. The multiplication factor to obtain a Rabin automaton is thus $2 \cdot 2 \cdot 2 \cdot 3 = 24$ and the state space is of the size $24 \cdot 1 \cdot (1 + 2^6) = 1560$.

7 Complexity

In this section, we summarize the theoretical complexity bounds we have obtained.

The traditional approach first translates the formula φ of length n into a non-deterministic automaton of size $\mathcal{O}(2^n)$. Then the determinization follows. The construction of Safra has the complexity $m^{\mathcal{O}(m)}$ where m is the size of the input automaton [Saf88]. This is in general optimal. The overall complexity is thus

$$2^{n \cdot \mathcal{O}(2^n)} = 2^{\mathcal{O}(2^{n+\log n})}$$

The recent lower bound for the whole LTL is $2^{2^{\mathcal{O}(n)}}$ [KRI10]. However, to be more precise, the example is of size less than $2^{\mathcal{O}(2^n)}$. Hence, there is a small gap. To the authors' best knowledge, there is no better upper bound when restricting to automata arising from LTL formulae or from the full (\mathbf{F}, \mathbf{G}) -fragment. (There are results on smaller fragments [AT04] though.) We tighten this gap slightly as shown below. Further, note that the number of Rabin pairs is $\mathcal{O}(m) = \mathcal{O}(2^n)$.

Our construction first produces a Muller automaton of size

$$\mathcal{O}(2^{2^{|\mathbb{T}|}} \cdot 2^{|Ap|}) = \mathcal{O}(2^{2^n+n}) \subseteq 2^{\mathcal{O}(2^n)}$$

which is strictly less than in the traditional approach. Moreover, as already discussed in Example 13, one can consider an “infinitary” fragment where every atomic proposition has in the syntactic tree both \mathbf{F} and \mathbf{G} as some ancestors. In this fragment, the state space of the Muller/generalized Rabin automaton is simply 2^{Ap} (when omitting the initial state) as for all $\alpha \subseteq Ap$, we have

$\text{succ}(\varphi, \alpha) = \varphi$. This is useful, since for e.g. fairness constraints our procedure yields exponentially smaller automaton.

Although the size of the Muller acceptance condition can be potentially exponentially larger than the state space, we have shown it can be compactly written as a disjunction of up to 2^n of conjunctions each of size at most n .

Moreover, using the intersection procedure we obtain a Rabin automaton with the upper bound on the state space

$$|\mathbb{F}|^{2^{|T|}} \cdot |Q| \in n^{2^n} \cdot 2^{\mathcal{O}(2^n)} = 2^{\mathcal{O}(\log n \cdot 2^n)} = 2^{\mathcal{O}(2^{n+\log \log n})} \subsetneq 2^{\mathcal{O}(2^{n+\log n})}$$

thus slightly improving the upper bound. Further, each conjunction is transformed into one pair, we are thus left with at most $2^{|T|} \in \mathcal{O}(2^n)$ Rabin pairs.

8 Experimental Results and Evaluation

We have implemented the construction of the state space of $\mathcal{A}(\varphi)$ described above. Further, Definition 15 then provides a way to compute the multiplication factor needed in order to get the Rabin automaton. We compare the sizes of this generalized Rabin automaton and Rabin automaton with the Rabin automaton produced by ltl2dstar. Ltl2dstar first calls an external translator from LTL to non-deterministic Büchi automata. In our experiments, it is LTL2BA [GO01] recommended by the authors of ltl2dstar. Then it performs Safra's determinization. Ltl2dstar implements several optimizations of Safra's construction. The optimizations shrink the state space by factor of 5 (saving 79.7% on average on the formulae considered here) to 10 (89.7% on random formulae) [KB06]. Our implementation does not perform any ad hoc optimization, since we want to evaluate whether the basic idea of the Safralss construction is already competitive. The only optimizations done are the following.

- Only the reachable part of the state space is generated.
 - Only atomic propositions relevant in each state are considered. In a state (χ, α) , a is not relevant if $\chi[a \mapsto \mathbf{tt}] \equiv \chi[a \mapsto \mathbf{ff}]$, i.e. if for every valuation, χ has the same value no matter which value a takes. For instance, let $Ap = \{a, b\}$ and consider $\chi = \mathfrak{U}(\mathbf{Fa}) = \mathbf{Fa} \vee a$. Then instead of having four copies (for $\emptyset, \{a\}, \{b\}, \{a, b\}$), there are only two for the sets of valuations $\{\emptyset, \{b\}\}$ and $\{\{a\}, \{a, b\}\}$. For its successor \mathbf{tt} , we only have one copy standing for the whole set $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$.
 - Definition 15 takes a disjunction over $I \in 2^{\mathbb{T}}$. If $I \subseteq I'$ but the set of states (χ, α) with $I \models_{\alpha} \chi$ and $I' \models_{\alpha} \chi$ are the same, it is enough to consider the disjunct for I only. E.g. for $\mathfrak{U}(\mathbf{G(Fa} \vee \mathbf{Fb)})$, we only consider I either $\{\mathbf{G(Fa} \vee \mathbf{Fb)}, \mathbf{Fa}\}$ or $\{\mathbf{G(Fa} \vee \mathbf{Fb)}, \mathbf{Fb}\}$, but not their union.
- This is an instance of a more general simplification. For a conjunction of pairs $(F_1, I_1) \wedge (F_2, I_2)$ with $I_1 \subseteq I_2$, there is a single equivalent condition $(F_1 \cup F_2, I_1)$.

Table II shows the results on formulae from BEEM (Benchmarks for Explicit Model checkers) [Pe107] and formulae from [SB00] on which ltl2dstar was originally tested [KB06]. In both cases, we only take formulae of the (\mathbf{F}, \mathbf{G}) -fragment.

In the first case this is 11 out of 20, in the second 12 out of 28. There is a slight overlap between the two sets. Further, we add conjunctions of strong fairness conditions and a few other formulae. For each formula φ , we give the number $|\text{states}(\varphi)|$ of distinct states w.r.t. the first (logical) component. The overall number of states of the Muller or generalized Rabin automaton follows. The respective runtimes are not listed as they were less than a second for all listed formulae, with the exception of the fifth formula from the bottom where it needed 3 minutes (here ltl2dstar needed more than one day to compute the Rabin automaton). In the column \mathcal{GR} -factor, we describe the complexity of the generalized Rabin condition, i.e. the number of copies of the state space that are created to obtain an equivalent Rabin automaton, whose size is thus bounded from above by the column Rabin. The last column states the size of the state space of the Rabin automaton generated by ltl2dstar using LTL2BA.

Table 1. Experimental comparison to ltl2dstar on formulae of [Pel07], [SB00], fairness constraints and some other examples of formulae of the “infinitary” fragment

Formula	states	Muller/GR	\mathcal{GR} -factor	Rabin	ltl2dstar
$\mathbf{G}(a \vee \mathbf{F}b)$	2	5	1	5	4
$\mathbf{F}\mathbf{G}a \vee \mathbf{F}\mathbf{G}b \vee \mathbf{G}\mathbf{F}c$	1	9	1	9	36
$\mathbf{F}(a \vee b)$	2	4	1	4	2
$\mathbf{G}\mathbf{F}(a \vee b)$	1	3	1	3	4
$\mathbf{G}(a \vee b \vee c)$	2	4	1	4	3
$\mathbf{G}(a \vee \mathbf{F}b)$	2	5	1	5	4
$\mathbf{G}(a \vee \mathbf{F}(b \vee c))$	2	5	1	5	4
$\mathbf{F}a \vee \mathbf{G}b$	3	7	1	7	5
$\mathbf{G}(a \vee \mathbf{F}(b \wedge c))$	2	5	1	5	4
$(\mathbf{F}\mathbf{G}a \vee \mathbf{G}\mathbf{F}b)$	1	5	1	5	12
$\mathbf{G}(a \vee b) \wedge \mathbf{G}\mathbf{F}(b \vee c)$	1	5	2	10	12
$(\mathbf{F}\mathbf{F}a \wedge \mathbf{G}\neg a) \vee (\mathbf{G}\mathbf{G}\neg a \wedge \mathbf{F}a)$	2	4	1	4	1
$(\mathbf{G}\mathbf{F}a) \wedge \mathbf{F}\mathbf{G}b$	1	5	1	5	7
$(\mathbf{G}\mathbf{F}a \wedge \mathbf{F}\mathbf{G}b) \vee (\mathbf{F}\mathbf{G}\neg a \wedge \neg b)$	1	5	1	5	14
$\mathbf{F}\mathbf{G}a \wedge \mathbf{G}\mathbf{F}a$	1	3	1	3	3
$\mathbf{G}(\mathbf{F}a \wedge \mathbf{F}b)$	1	5	2	10	5
$\mathbf{F}a \wedge \mathbf{F}b$	4	8	1	8	4
$(\mathbf{G}(b \vee \mathbf{G}Fa) \wedge \mathbf{G}(c \vee \mathbf{G}\mathbf{F}\neg a)) \vee \mathbf{G}b \vee \mathbf{G}c$	4	18	2	36	26
$(\mathbf{G}(b \vee \mathbf{F}Ga) \wedge \mathbf{G}(c \vee \mathbf{F}\mathbf{G}\neg a)) \vee \mathbf{G}b \vee \mathbf{G}c$	4	18	1	18	29
$(\mathbf{F}(b \wedge \mathbf{F}Ga) \vee \mathbf{F}(c \wedge \mathbf{F}\mathbf{G}\neg a)) \wedge \mathbf{F}b \wedge \mathbf{F}c$	4	18	1	18	8
$(\mathbf{F}(b \wedge \mathbf{G}Fa) \vee \mathbf{F}(c \wedge \mathbf{G}\mathbf{F}\neg a)) \wedge \mathbf{F}b \wedge \mathbf{F}c$	4	18	1	18	45
$(\mathbf{F}\mathbf{G}a \vee \mathbf{G}\mathbf{F}b)$	1	5	1	5	12
$(\mathbf{F}\mathbf{G}a \vee \mathbf{G}\mathbf{F}b) \wedge (\mathbf{F}\mathbf{G}c \vee \mathbf{G}\mathbf{F}d)$	1	17	2	34	17527
$\bigwedge_{i=1}^3 (\mathbf{G}\mathbf{F}a_i \rightarrow \mathbf{G}\mathbf{F}b_i)$	1	65	24	1 560	1 304 706
$(\bigwedge_{i=1}^5 \mathbf{G}\mathbf{F}a_i) \rightarrow \mathbf{G}\mathbf{F}b$	1	65	1	65	972
$\mathbf{G}\mathbf{F}(\mathbf{F}a \mathbf{G}\mathbf{F}b \mathbf{F}\mathbf{G}(a \vee b))$	1	5	1	5	159
$\mathbf{F}\mathbf{G}(\mathbf{F}a \vee \mathbf{G}\mathbf{F}b \vee \mathbf{F}\mathbf{G}(a \vee b))$	1	5	1	5	2918
$\mathbf{F}\mathbf{G}(\mathbf{F}a \vee \mathbf{G}\mathbf{F}b \vee \mathbf{F}\mathbf{G}(a \vee b) \vee \mathbf{F}\mathbf{G}b)$	1	5	1	5	4516

While the advantages of our approach over the general determinization are clear for the infinitary fragment, there seem to be some drawbacks when “finitary” behaviour is present, i.e. behaviour that can be satisfied or disproved after finitely many steps. The reason and the patch for this are the following. Consider the formula $\mathbf{F}a$ and its automaton from Example 5. Observe that one can easily collapse the automaton to the size of only 2. The problem is that some states such as $\langle a \vee \mathbf{X}\mathbf{F}a, \{a\} \rangle$ are only “passed through” and are equivalent to some of their successors, here $\langle \mathbf{t}\mathbf{t}, \{a\} \rangle$. However, we may safely perform the following collapse. Whenever two states $(\chi, \alpha), (\chi', \alpha)$ satisfy that $\chi[\alpha \mapsto \mathbf{t}\mathbf{t}, Ap \setminus \alpha \mapsto \mathbf{f}\mathbf{f}]$ is propositionally equivalent to $\chi'[\alpha \mapsto \mathbf{t}\mathbf{t}, Ap \setminus \alpha \mapsto \mathbf{f}\mathbf{f}]$ we may safely merge the states as they have the same properties: they are bisimilar with the same set of atomic propositions satisfied. Using these optimizations, e.g. the automaton for $\mathbf{F}a \wedge \mathbf{F}b$ has size 4 as the one produced by ltl2dstar.

Next important observation is that the blow-up from generalized Rabin to Rabin automaton (see the column \mathcal{GR} -factor) corresponds to the number of elements of \mathbb{F} that have a descendant or an ancestor in \mathbb{G} and are combined with conjunction. This follows directly from the transformation described in Section 6 and is illustrated in the table.

Thus, we may conclude that our approach is competitive to the determinization approach and for some classes of useful properties such as fairness constraints or generally the infinitary properties it shows significant advantages. Firstly, the state space of the Rabin automaton is noticeably smaller. Secondly, compact generalized Rabin automata tend to be small even for more complex formulae. Thirdly, the state spaces of our automata have a clear structure to be exploited for further possible optimizations, which is more difficult in the case of determinization. In short, the state space is less “messy”.

9 Discussion on Extensions

Our approach seems to be extensible to the $(\mathbf{X}, \mathbf{F}, \mathbf{G})$ -fragment. In this setting, instead of remembering the one-step history one needs to remember n last steps (or have a n -step look-ahead) in order to deal with formulae such as $\mathbf{GF}(a \wedge \mathbf{X}b)$. Indeed, the acceptance condition requires to visit infinitely often a state provably satisfying $a \wedge \mathbf{X}b$. This can be done by remembering the last n symbols read, where n can be chosen to be the nesting depth of \mathbf{X} s. We have not presented this extension mainly for the sake of clarity of the construction.

Further, one could handle the positive (\mathbf{X}, \mathbf{U}) -fragment, where only atomic propositions may be negated as defined above. These formulae are purely “finitary” and the logical component of the state space is sufficient. Indeed, the automaton simply accepts if and only if $\mathbf{t}\mathbf{t}$ is reached and there is no need to check any formulae that we had committed to.

For the (\mathbf{U}, \mathbf{G}) -fragment or the whole LTL, our approach would need to be significantly enriched as the state space (and last n symbols read) is not sufficient to keep enough information to decide whether a run ρ is accepting only based on $\text{Inf}(\rho)$. Indeed, consider a formula $\varphi = \mathbf{GF}(a \wedge b\mathbf{U}c)$. Then reading $\{a, b\}$ results

in the requirement $\mathbf{GF}(a \wedge b \mathbf{U} c) \wedge (\mathbf{F}(a \wedge b \mathbf{U} c) \vee (b \mathbf{U} c))$ which is, however, temporally equivalent to φ (their unfolds are propositionally equivalent). Thus, runs on $(\{a, b\}\{c\}\emptyset)^\omega$ and $(\{a, b\}\emptyset\{c\})^\omega$ have the same set of infinitely often visited states. Hence, the order of visiting the states matters and one needs the history. However, words such as $(\{a, b\}\{b\}^n\{c\})^\omega$ vs. $(\{b\}^n\{c\})^\omega$ show that more complicated structure is needed than last n letters. The conjecture that this approach is extensible to the whole LTL is left open and considered for future work.

10 Conclusions

We have shown a direct translation of the LTL fragment with operators \mathbf{F} and \mathbf{G} to deterministic automata. This translation has several advantages compared to the traditional way that goes via non-deterministic Büchi automata and then performs determinization. First of all, in our opinion it is a lot simpler than the determinization and its various non-trivial optimizations. Secondly, the state space has a clear logical structure. Therefore, any work with the automata or further optimizations seem to be conceptually easier. Moreover, many optimizations are actually done by the logic itself. Indeed, logical equivalence of the formulae helps to shrink the state space with no further effort. In a sense, the logical part of a state contains precisely the information that the semantics of LTL dictates, see Proposition 9. Thirdly, the state space is—according to the experiments—not much bigger even when compared to already optimized determinization. Moreover, very often it is considerably smaller, especially for the “infinitary” formulae; in particular, for fairness conditions. Furthermore, we have also given a very compact deterministic ω -automaton with a small and in our opinion reasonably simple generalized Rabin acceptance condition.

Although we presented a possible direction to extend the approach to the whole LTL, we leave this problem open and will focus on this in future work. Further, since only the obvious optimizations mentioned in Section 8 have been implemented so far, there is space for further performance improvements in this new approach.

Acknowledgement. Thanks to Andreas Gaiser for pointing out to us that ltl2dstar constructs surprisingly large automata for fairness constraints and the anonymous reviewers for their valuable comments.

References

- [AT04] Alur, R., La Torre, S.: Deterministic generators and games for LTL fragments. ACM Trans. Comput. Log. 5(1), 1–25 (2004)
- [BK08] Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)
- [GO01] Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001), Tool accessible at
<http://www.lsv.ens-cachan.fr/>

- [JB06] Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE Computer Society (2006)
- [JGWB07] Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A Tool for Property Synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
- [KB06] Klein, J., Baier, C.: Experiments with deterministic omega-automata for formulas of linear temporal logic. *Theor. Comput. Sci.* 363(2), 182–195 (2006)
- [KB07] Klein, J., Baier, C.: On-the-Fly Stuttering in the Construction of Deterministic ω -Automata. In: Holub, J., Žd’árek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 51–61. Springer, Heidelberg (2007)
- [Kle] Klein, J.: ltl2dstar - LTL to deterministic Streett and Rabin automata, <http://www.ltl2dstar.de/>
- [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)
- [KPV06] Kupferman, O., Piterman, N., Vardi, M.Y.: Safrales Compositional Synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)
- [KR10] Kupferman, O., Rosenberg, A.: The Blow-Up in Translating LTL to Deterministic Automata. In: van der Meyden, R., Smaus, J.-G. (eds.) MoChArt 2010. LNCS, vol. 6572, pp. 85–94. Springer, Heidelberg (2011)
- [Kup12] Kupferman, O.: Recent Challenges and Ideas in Temporal Synthesis. In: Bieliková, M., Friedrich, G., Gottlob, G., Katzenbeisser, S., Turán, G. (eds.) SOFSEM 2012. LNCS, vol. 7147, pp. 88–98. Springer, Heidelberg (2012)
- [KV05] Kupferman, O., Vardi, M.Y.: Safrales decision procedures. In: FOCS, pp. 531–542. IEEE Computer Society (2005)
- [MS08] Morgenstern, A., Schneider, K.: From LTL to Symbolically Represented Deterministic Automata. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 279–293. Springer, Heidelberg (2008)
- [Pel07] Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
- [Pit06] Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: LICS, pp. 255–264. IEEE Computer Society (2006)
- [Pnu77] Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
- [PPS06] 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)
- [PR88] Pnueli, A., Rosner, R.: A Framework for the Synthesis of Reactive Modules. In: Vogt, F.H. (ed.) CONCURRENCY 1988. LNCS, vol. 335, pp. 4–17. Springer, Heidelberg (1988)
- [Saf88] Safra, S.: On the complexity of ω -automata. In: FOCS, pp. 319–327. IEEE Computer Society (1988)
- [SB00] Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)

Efficient Controller Synthesis for Consumption Games with Multiple Resource Types

Tomáš Brázdil^{1,*}, Krishnendu Chatterjee^{2,*}, Antonín Kučera^{1,*}, and Petr Novotný^{1,*}

¹ Faculty of Informatics, Masaryk University, Brno, Czech Republic

{brazdil,kucera,xnovot18}@fi.muni.cz

² IST Austria, Klosterneuburg, Austria

krish.chat@gmail.com

Abstract. We introduce *consumption games*, a model for discrete interactive system with multiple resources that are consumed or reloaded independently. More precisely, a consumption game is a finite-state graph where each transition is labeled by a vector of resource updates, where every update is a non-positive number or ω . The ω updates model the reloading of a given resource. Each vertex belongs either to player \square or player \diamond , where the aim of player \square is to play so that the resources are never exhausted. We consider several natural algorithmic problems about consumption games, and show that although these problems are computationally hard in general, they are solvable in polynomial time for every fixed number of resource types (i.e., the dimension of the update vectors) and bounded resource updates.

1 Introduction

In this paper we introduce *consumption games*, a model for discrete interactive systems with multiple resources that can be consumed and reloaded independently. We show that consumption games, despite their rich modeling power, still admit efficient algorithmic analysis for a “small” number of resource types. This property distinguishes consumption games from other related models, such as games over vector addition systems or multi-energy games (see below), that are notoriously intractable.

Roughly speaking, a consumption game is a finite-state directed graph where each state belongs either to player \square (controller) or player \diamond (environment). Every transition $s \rightarrow t$ is labeled by a d -dimensional vector δ such that each component $\delta(i)$ is a non-positive integer (encoded in *binary*) or ω . Intuitively, if $\delta(i) = -n$, then the current load of the i -th resource is decreased by n while performing $s \rightarrow t$, and if $\delta(i) = \omega$, then the i -th resource can be “reloaded” to an arbitrarily high value greater than or equal to the current load. A *configuration* of a consumption game is determined by the current control state and the current load of all resources, which is a d -dimensional vector of positive integers. A play of a consumption game is initiated in some state and some

* Tomáš Brázdil, Antonín Kučera, and Petr Novotný are supported by the Czech Science Foundation, grant No. P202/10/1469. Krishnendu Chatterjee is supported by the FWF (Austrian Science Fund) NFN Grant No S11407-N23 (RiSE) and ERC Start grant (279307: Graph Games).

initial load of resources. The aim of player \square is to play *safely*, i.e., select transitions in his states so that the vector of current resource loads stays positive in every component (i.e., the resources are never exhausted). Player \diamond aims at the opposite.

The resources may correspond to fuel, electricity, money, or even more abstract entities such as time or patience. To get a better intuition behind consumption games and the abstract problems studied in this paper, let us discuss one particular example in greater detail.

The public transport company of Brno city¹ maintains the network of public trams, buses, trolleybuses, and boats. Due to the frequent failures and breakdowns in electrical wiring, rails, railroad switches, and the transport vehicles, the company has several emergency teams which travel from one accident to another according to the directives received from the central supervisory office. Recently, the company was considering the possibility of replacing their old diesel vans by new cars equipped with more ecological natural gas engines. The problem is that these cars have smaller range and can be tanked only at selected gas stations. So, it is not clear whether the cars are usable at all, i.e., whether they can always visit a gas station on time regardless where and when an accident happens, and what are the time delays caused by detours to gas stations. Now we indicate how to construct the associated consumption game model and how to rephrase the above questions formally.

We start with a standard graph G representing the city road network, i.e., the nodes of G correspond to distinguished locations (such as crossings) and the edges correspond to the connecting roads. Then we identify the nodes corresponding to gas stations that sell natural gas, and to each edge (road) we assign two negative numbers corresponding to the expected time and fuel needed to pass the road. Every morning, a car leaves a central garage (where it is fully tanked) and returns to the same place in the evening. The maximal number of accidents serviced per day can be safely overestimated by 12. Our consumption game C has two resource types modeling the fuel and time in the expected way. The fuel is consumed by passing a transition (road), and can be reloaded by the outgoing transitions of gas stations. The time is also consumed by passing the roads, and the only node where it can be reloaded is the central garage, but only after completing the 12 jobs. In the states of C we remember the current job number (from 1 to 12) and the current target node. At the beginning, and also after visiting the current target node, the next target node is selected by player \diamond . Technically, the current target node belongs to player \diamond , and there is a transition for every (potential) next target node. Performing such a transition does not consume the resources, but the information about the next target node is stored in the chosen state, job index is increased, and the control over the play is given back to player \square who models the driver. This goes on until the job index reaches 12. Then, player \diamond makes no further choice, but it is possible to reload the time resource at the node corresponding to the central garage, and hence player \square aims at returning to this place as quickly as possible (without running out of gas). Note that C has about $12 \cdot n^2$ states, where n is the number of states of G .

The question whether the new cars are usable at all can now be formalized as follows: *Is there is safe strategy for player \square in the initial configuration such that the fuel resource is never reloaded to a value which is higher than the tank capacity of the car?*

¹ DPMB, Dopravní Podnik Města Brna.

In the initial configuration, the fuel resource is initialized to 1 because it can be immediately reloaded in the central garage, and the time resource is initialized to a “sufficiently high value” which is efficiently computable due to the *finite reload property* formulated in Corollary 7. Similarly, the extra time delays caused by detours to gas stations can be estimated by computing the *minimal initial credit* for the time resource, i.e., the minimal initial value sufficient for performing a safe strategy, and comparing this number with the minimal initial credit for the time resource in a simplified consumption game where the fuel is not consumed at all (this corresponds to an ideal “infinite tank capacity”). Similarly, one could also analyze the extra fuel costs, or model the consumption of the material needed to perform the repairs, and many other aspects.

An important point of the above example is that the number of resources is relatively small, but the number of states is large. This motivates the study of *parameterized complexity* of basic decision/optimization problems for consumption games, where the parameters are the following:

- d , the number of resources (or *dimension*);
- ℓ , the maximal finite $|\delta(i)|$ such that $1 \leq i \leq d$ and δ is a label of some transition.

Main Results. For every state s of a consumption game C , we consider the following sets of vectors (see Section 2 for precise definitions):

- $\text{Safe}(s)$ consists of all vectors α of positive integers such that player \square has a safe strategy in the configuration (s, α) . That is, $\text{Safe}(s)$ consists of all vectors describing a sufficient *initial* load of all resources needed to perform a safe strategy.
- $\text{Cover}(s)$ consists of all vectors α of positive integers such that player \square has a safe strategy σ in the configuration (s, α) such that for every strategy π for player \diamond and every configuration (t, β) visited during the play determined by σ and π we have that $\beta \leq \alpha$. Note that physical resources (such as fuel, water, electricity, etc.) are stored in devices with finite capacity (tanks, batteries, etc.), and hence it is important to know what capacities of these devices are sufficient for performing a safe strategy. These sufficient capacities correspond to the vectors of $\text{Cover}(s)$.

Clearly, both $\text{Safe}(s)$ and $\text{Cover}(s)$ are upwards closed with respect to component-wise ordering. Hence, these sets are fully determined by their *finite* sets of minimal elements. In this paper we aim at answering the very basic algorithmic problems about $\text{Safe}(s)$ and $\text{Cover}(s)$, which are the following:

- (A) *Emptiness.* For a given state s , decide whether $\text{Safe}(s) = \emptyset$ (or $\text{Cover}(s) = \emptyset$).
- (B) *Membership.* For a given state s and a vector α , decide whether $\alpha \in \text{Safe}(s)$ (or $\alpha \in \text{Cover}(s)$). Further, decide whether α is a *minimal* vector of $\text{Safe}(s)$ (or $\text{Cover}(s)$).
- (C) *Compute the set of minimal vectors of $\text{Safe}(s)$ (or $\text{Cover}(s)$).*

Note that these problems subsume the questions of our motivating example. We show that *all of these problems are computationally hard*, but solvable in *polynomial time* for every fixed choice of the parameters d and ℓ introduced above. Since the degree of the bounding polynomial increases with the size of the parameters, we do *not* provide fixed-parameter tractability results in the usual sense of parameterized complexity (as it is mentioned in Section 3, this would imply a solution to a long-standing open problem in study of graph games). Still, these results clearly show that for “small” parameter

values, the above problems *are* practically solvable even if the underlying graph of C is very large. More precisely, we show the following for game graphs with n states:

- The emptiness problems for $\text{Safe}(s)$ and $\text{Cover}(s)$ are **coNP**-complete, and solvable in $O(d! \cdot n^{d+1})$ time.
- The membership problems for $\text{Safe}(s)$ and $\text{Cover}(s)$ are **PSPACE**-hard and solvable in time $|\alpha| \cdot (d \cdot \ell \cdot n)^{O(d)}$ and $O(\Lambda^2 \cdot n^2)$, respectively, where $|\alpha|$ is the encoding size of α and $\Lambda = \prod_{i=1}^d \alpha(i)$.
- The set of minimal elements of $\text{Safe}(s)$ and $\text{Cover}(s)$ is computable in time $(d \cdot \ell \cdot n)^{O(d)}$ and $(d \cdot \ell \cdot n)^{O(d \cdot d!)}$, respectively.

Then, in Section 4, we show that the complexity of some of the above problems can be substantially improved for two natural subclasses of *one-player* and *decreasing* consumption games by employing special methods. A consumption game is *one-player* if all states are controlled by player \square , and *decreasing* if every resource is either reloaded or decreased along every cycle in the graph of C . For example, the game constructed in our motivating example is decreasing, and we give a motivating example for one-player consumption games in Section 4. In particular, we prove that

- the emptiness problem for $\text{Safe}(s)$ and $\text{Cover}(s)$ is solvable in polynomial time both for one-player and decreasing consumption games;
- the membership problem for $\text{Safe}(s)$ is **PSPACE**-complete (resp. **NP**-complete) for decreasing consumption games (resp. one-player consumption games).
- Furthermore, for both these subclasses we present algorithms to compute the minimal elements of $\text{Safe}(s)$ by a reduction to *minimum multi-distance reachability* problem, and solving the minimum multi-distance reachability problem on game graphs. Though these algorithms do not improve the worst case complexity over general consumption games, they are iterative and potentially terminate much earlier (we refer to Section 4.3 and Section 4.4 for details).

Related Work. Our model of consumption games is related but incomparable to *energy games* studied in the literature. In energy games both positive and non-positive weights are allowed, but in contrast to consumption games there are no ω -weights. Energy games with single resource type were introduced in [5], and it was shown that the minimal initial credit problem (and also the membership problem for $\text{Safe}(s)$) can be solved in exponential time. Further, it follows from the results of [5] that the emptiness problem for $\text{Safe}(s)$, which was shown to be equivalent to two-player mean-payoff games [2], lies in $\mathbf{NP} \cap \mathbf{coNP}$.

Games over extended vector addition systems with states (eVASS games), where the weights in transition labels are in $\{-1, 0, 1, \omega\}$, were introduced and studied in [4]. In [4], it was shown that the question whether player \square has a safe strategy in a given configuration is decidable, and the winning region of player \square is computable in $(d - 1)\text{-EXPTIME}$, where d is the eVASS dimension, and hence the provided solution is impractical even for very small d 's. A closely related model of energy games with multiple resource types (or multi-energy games) was considered in [7]. The minimal initial credit problem (and also the membership problem for $\text{Safe}(s)$) for multi-energy games can be reduced to the corresponding problem over eVASS games with an exponential reduction to encode the integer weights into weights $\{-1, 0, 1\}$. Thus the minimal initial credit problem can be solved in $d\text{-EXPTIME}$, and the membership problem is

EXPSPACE-hard (the hardness follows from the classical result of Lipton [12]). The emptiness problem for $\text{Safe}(s)$ is **coNP**-complete for multi-energy games [7]. Thus the complexity of the membership and the minimal initial credit problem for consumption games is much better (it is in **EXPTIME** and **PSPACE**-hard and can be solved in polynomial time for every fixed choice of the parameters) as compared to eVASS games or multi-energy games (**EXPSPACE**-hard and can be solved in d -**EXPTIME**). For eVASS games with fixed dimensions, the problem can be solved in polynomial time for $d = 2$ (see [6]), and it is open whether the complexity can be improved for other constants. Moreover, for the important subclasses of one-player and decreasing consumption games we show much better bounds (polynomial time algorithms for emptiness and optimal complexity bounds for membership in $\text{Safe}(s)$).

The paper is organized as follows. After presenting necessary definitions in Section 2, we present our solution to the three algorithmic problems (A)-(C) for general consumption games in Section 3. In Section 4, we concentrate on the two subclasses of decreasing and one-player consumption games and give optimized solutions to some of these problems. Finally, in Section 5 we give a short list of open problems which, in our opinion, address some of the fundamental properties of consumption games that deserve further attention. Due to the lack of space, the proofs are omitted. They can be found in the full version of this paper [3].

2 Definitions

In this paper, the set of all integers is denoted by \mathbb{Z} . For a given operator $\bowtie \in \{>, <, \leq, \geq\}$, we use $\mathbb{Z}_{\bowtie 0}$ to denote the set $\{i \in \mathbb{Z} \mid i \bowtie 0\}$, and $\mathbb{Z}_{\bowtie 0}^\omega$ to denote the set $\mathbb{Z}_{\bowtie 0} \cup \{\omega\}$, where $\omega \notin \mathbb{Z}$ is a special symbol representing an “infinite amount” with the usual conventions (in particular, $c + \omega = \omega + c = \omega$ and $c < \omega$ for every $c \in \mathbb{Z}$). For example, $\mathbb{Z}_{< 0}$ is the set of all negative integers, and $\mathbb{Z}_{< 0}^\omega$ is the set $\mathbb{Z}_{< 0} \cup \{\omega\}$. We use Greek letters α, β, \dots to denote vectors over $\mathbb{Z}_{\bowtie 0}$ or $\mathbb{Z}_{\bowtie 0}^\omega$, and $\mathbf{0}$ to denote the vector of zeros. The i -th component of a given α is denoted by $\alpha(i)$. The standard component-wise ordering over vectors is denoted by \leq , and we also write $\alpha < \beta$ to indicate that $\alpha(i) < \beta(i)$ for every i .

Let M be a finite or countably infinite alphabet. A *word* over M is a finite or infinite sequence of elements of M . The empty word is denoted by ε , and the set of all finite words over M is denoted by M^* . Sometimes we also use M^+ to denote the set $M^* \setminus \{\varepsilon\}$. The length of a given word w is denoted by $\text{len}(w)$, where $\text{len}(\varepsilon) = 0$ and the length of an infinite word is ∞ . The individual letters in a word w are denoted by $w(0), w(1), \dots$, and for every infinite word w and every $i \geq 0$ we use w_i to denote the infinite word $w(i), w(i+1), \dots$

A *transition system* is a pair $\mathcal{T} = (V, \rightarrow)$, where V is a finite or countably infinite set of *vertices* and $\rightarrow \subseteq V \times V$ a *transition relation* such that for every $v \in V$ there is at least one outgoing transition (i.e., a transition of the form $v \rightarrow u$). A *path* in \mathcal{T} is a finite or infinite word w over V such that $w(i) \rightarrow w(i+1)$ for every $0 \leq i < \text{len}(w)$. We call a finite path a *history* and infinite path a *run*. The sets of all finite paths and all runs in \mathcal{T} are denoted by $\text{FPath}(\mathcal{T})$ and $\text{Run}(\mathcal{T})$, respectively.

Definition 1. A (2-player) game is a triple $G = (V, \mapsto, (V_\square, V_\diamond))$ where (V, \mapsto) is a transition system and (V_\square, V_\diamond) is a partition of V . If $V_\diamond = \emptyset$, then G is a 1-player game.

A game G is played by two players, \square and \diamond , who select transitions in the vertices of V_{\square} and V_{\diamond} , respectively. Let $\odot \in \{\square, \diamond\}$. A *strategy* for player \odot is a function which to each $wv \in V^*V_{\odot}$ assigns a state $v' \in V$ such that $v \mapsto v'$. The sets of all strategies for player \square and player \diamond are denoted by Σ_G and Π_G (or just by Σ and Π if G is understood), respectively. We say that a strategy τ is *memoryless* if $\tau(wv)$ depends just on the last state v , for every $w \in V^*$. Strategies that are not necessarily memoryless are called *history-dependent*. Note that every initial vertex v and every pair of strategies $(\sigma, \pi) \in \Sigma \times \Pi$ determine a unique infinite path in G initiated in v , which is called a *play* and denoted by $Play_{\sigma, \pi}(v)$.

Definition 2. Let $d \geq 1$. A consumption game of dimension d is a tuple $C = (S, E, (S_{\square}, S_{\diamond}), L)$ where S is a finite set of states, (S, E) is a transition system, $(S_{\square}, S_{\diamond})$ is a partition of S , and L is labelling which to every $(s, t) \in E$ assigns a vector $\delta = (\delta(1), \dots, \delta(d))$ such that $\delta(i) \in \mathbb{Z}_{\leq 0}^{\omega}$ for every $1 \leq i \leq d$. If $s \in S_{\diamond}$, we require that $\delta(i) \neq \omega$ for all $1 \leq i \leq d$. We write $s \xrightarrow{\delta} t$ to indicate that $(s, t) \in E$ and $L(s, t) = \delta$.

We say that C is one-player if $S_{\diamond} = \emptyset$, and decreasing if for every $n \geq 1$, every $1 \leq i \leq d$, and every path $s_0 \xrightarrow{\delta_1} s_1 \xrightarrow{\delta_2} \dots \xrightarrow{\delta_n} s_n$ such that $s_0 = s_n$, there is some $j \leq n$ where $\delta_j(i) \neq 0$.

Intuitively, if $s \xrightarrow{\delta} t$, then the system modeled by C can move from the state s to the state t so that its resources are consumed/reloaded according to δ . More precisely, if $\delta(i) \leq 0$, then the current load of resource i is decreased by $|\delta(i)|$, and if $\delta(i) = \omega$, then the resource i can be reloaded to an arbitrarily high positive value larger than or equal to the current load. The aim of player \square is to play so that the resources are never exhausted, i.e., the vector of current loads stays positive in every component. The aim of player \diamond is to achieve the opposite.

The above intuition is formally captured by defining the associated infinite-state game G_C for C . The vertices of G_C are *configurations* of C , i.e., the elements of $S \times \mathbb{Z}_{>0}^d$ together with a special configuration F (which stands for “fail”). The transition relation \mapsto of G_C is determined as follows:

- $F \mapsto F$.
- For every configuration (s, α) and every transition $s \xrightarrow{\delta} t$ of C such that $\alpha(i) + \delta(i) > 0$ for all $1 \leq i \leq d$, there is a transition $(s, \alpha) \mapsto (t, \alpha + \gamma)$ for every $\gamma \in \mathbb{Z}^d$ such that
 - $\gamma(i) = \delta(i)$ for every $1 \leq i \leq d$ where $\delta(i) \neq \omega$;
 - $\gamma(i) \geq 0$ for every $1 \leq i \leq d$ where $\delta(i) = \omega$.
- If (s, α) is a configuration and $s \xrightarrow{\delta} t$ a transition of C such that $\alpha(i) + \delta(i) \leq 0$ for some $1 \leq i \leq d$, then there is a transition $(s, \alpha) \mapsto F$.
- There are no other transitions.

A strategy σ for player \square in G_C is *safe* in a configuration (s, α) iff for every strategy π for player \diamond we have that $Play_{\sigma, \pi}(s, \alpha)$ does not visit the configuration F . For every $s \in S$, we use

- $Safe(s)$ to denote the set of all $\alpha \in \mathbb{Z}_{>0}^d$ such that player \square has a safe strategy in (s, α) ;
- $Cover(s)$ to denote the set of all $\alpha \in \mathbb{Z}_{>0}^d$ such that player \square has a safe strategy σ in (s, α) such that for every strategy π for player \diamond and every configuration (t, β) visited by $Play_{\sigma, \pi}(s, \alpha)$ we have that $\beta \leq \alpha$.

If $\alpha \in \text{Safe}(s)$, we say that α is *safe in* s , and if $\alpha \in \text{Cover}(s)$, we say that α *covers* s . Obviously, $\text{Cover}(s) \subseteq \text{Safe}(s)$, and both $\text{Safe}(s)$ and $\text{Cover}(s)$ are upwards closed w.r.t. component-wise ordering (i.e., if $\alpha \in \text{Safe}(s)$ and $\alpha \leq \alpha'$, then $\alpha' \in \text{Safe}(s)$). This means that $\text{Safe}(s)$ and $\text{Cover}(s)$ are fully described by its finitely many minimal elements.

Intuitively, $\text{Safe}(s)$ consists of all vectors describing a sufficiently large *initial* amount of all resources needed to perform a safe strategy. Note that during a play, the resources can be reloaded to values that are larger than the initial one. Since physical resources are stored in “tanks” with finite capacity, we need to know what *capacities* of these tanks are sufficient for performing a safe strategy. These sufficient capacities are encoded by the vectors of $\text{Cover}(s)$.

3 Algorithms for General Consumption Games

In this section we present a general solution for the three algorithmic problems (A)-(C) given in Section I.

We start by a simple observation that connects the study of consumption games to a more mature theory of Streett games. A Streett game is a tuple $\mathcal{S} = (V, \mapsto, (V_\square, V_\diamond), \mathcal{A})$, where $(V, \mapsto, (V_\square, V_\diamond))$ is a 2-player game with finitely many vertices, and $\mathcal{A} = \{(G_1, R_1), \dots, (G_m, R_m)\}$, where $m \geq 1$ and $G_i, R_i \subseteq \mapsto$ for all $1 \leq i \leq m$, is a Streett (or strong fairness) winning condition (for technical convenience, we consider G_i, R_i as subsets of edges rather than vertices). For an infinite path w in \mathcal{S} , let $\inf(w)$ be the set of all edges that are executed infinitely often along w . We say that w *satisfies* \mathcal{A} iff $\inf(w) \cap G_i \neq \emptyset$ implies $\inf(w) \cap R_i \neq \emptyset$ for every $1 \leq i \leq m$. A strategy $\sigma \in \Sigma_{\mathcal{S}}$ is *winning* in $v \in V$ if for every $\pi \in \Pi_{\mathcal{S}}$ we have that $\text{Play}_{\sigma, \pi}(v)$ satisfies \mathcal{A} . The problem whether player \square has a winning strategy in a vertex $v \in V$ is **coNP**-complete [Q], and the problem can be solved in $O(m! \cdot |V|^{m+1})$ time [I3].

For the rest of this section, we fix a consumption game $C = (S, E, (S_\square, S_\diamond), L)$ of dimension d , and we use ℓ to denote the maximal finite $|\delta(i)|$ such that $1 \leq i \leq d$ and δ is a label of some transition.

Lemma 3. *Let $\mathcal{S}_C = (S, E, (S_\square, S_\diamond), \mathcal{A})$ be a Streett game where $\mathcal{A} = \{(G_1, R_1), \dots, (G_d, R_d)\}$, $G_i = \{(s, t) \in E \mid L(s, t)(i) < 0\}$, and $R_i = \{(s, t) \in E \mid L(s, t)(i) = \omega\}$ for every $1 \leq i \leq d$. Then for every $s \in S$ the following assertions hold:*

1. *If $\text{Safe}(s) \neq \emptyset$, then player \square has a winning strategy in s in the Streett game \mathcal{S}_C .*
2. *If player \square has a winning strategy in s in the Streett game \mathcal{S}_C , then $(d! \cdot |S| \cdot \ell + 1, \dots, d! \cdot |S| \cdot \ell + 1) \in \text{Safe}(s) \cap \text{Cover}(s)$.*

An immediate consequence of Lemma 3 is that $\text{Safe}(s) = \emptyset$ iff $\text{Cover}(s) = \emptyset$. Our next lemma shows that the existence of a winning strategy in Streett games is polynomially reducible to the problem whether $\text{Safe}(s) = \emptyset$ in consumption games.

Lemma 4. *Let $\mathcal{S} = (V, \mapsto, (V_\square, V_\diamond), \mathcal{A})$ be a Streett game where $\mathcal{A} = \{(G_1, R_1), \dots, (G_m, R_m)\}$. Let $C_{\mathcal{S}} = (V, \mapsto, (V_\square, V_\diamond), L)$ be a consumption game of dimension m where $L(u, v)(i)$ is either -1 , ω , or 0 , depending on whether $(u, v) \in G_i$, $(u, v) \in R_i$, or $(u, v) \notin G_i \cup R_i$, respectively. Then for every $v \in V$ we have that player \square has a winning strategy in v (in \mathcal{S}) iff $\text{Safe}(v) \neq \emptyset$ (in $C_{\mathcal{S}}$).*

A direct consequence of Lemma 3 and Lemma 4 is the following:

Theorem 5. *The emptiness problems for $\text{Safe}(s)$ and $\text{Cover}(s)$ are coNP -complete and solvable in $O(d! \cdot |S|^{d+1})$ time.*

Also observe that if managed to prove that the emptiness problem for $\text{Safe}(s)$ or $\text{Cover}(s)$ is fixed-parameter tractable in d for consumption games where ℓ is equal to one (i.e., if we proved that the problem is solvable in time $F(d) \cdot n^{O(1)}$ where n is the size of the game and F a computable function), then due to Lemma 4 we would immediately obtain that the problem whether player \square has a winning strategy in a given Streett game is also fixed-parameter tractable. That is, we would obtain a solution to one of the long-standing open problems of algorithmic study of graph games.

Now we show how to compute the set of minimal elements of $\text{Safe}(s)$. A key observation is the following lemma whose proof is non-trivial.

Lemma 6. *For every $s \in S$ and every minimal $\alpha \in \text{Safe}(s)$ we have that $\alpha(i) \leq d \cdot \ell \cdot |S|$ for every $1 \leq i \leq d$.*

Observe that Lemma 6 does not follow from Lemma 3(2.). Apart from Lemma 6 providing better bound, Lemma 3(2.) only says that if all resources are loaded enough, then there is a safe strategy. However, we aim at proving a substantially stronger result saying that no resource needs to be reloaded to more than $d \cdot \ell \cdot |S|$ regardless how large is the current load of other resources.

Intuitively, Lemma 6 is obtained by a somewhat tricky inductive argument where we first consider all resources as being “sufficiently large” and then bound the components one by one. Since a similar technique is also used to compute the minimal elements of $\text{Cover}(s)$, we briefly introduce the main underlying notions and ideas.

An *abstract load vector* μ is an element of $(\mathbb{Z}_{>0}^\omega)^d$. The *precision* of μ is the number of components different from ω . The standard componentwise ordering is extended also to abstract load vectors by stipulating that $c < \omega$ for every $c \in \mathbb{Z}$. Given an abstract load vector μ and a vector $\alpha \in (\mathbb{Z}_{>0})^d$, we say that α *matches* μ if $\alpha(j) = \mu(j)$ for all $1 \leq j \leq d$ such that $\mu(j) \neq \omega$. Finally, we say that μ is *compatible* with $\text{Safe}(s)$ (or $\text{Cover}(s)$) if there is some $\alpha \in \text{Safe}(s)$ (or $\alpha \in \text{Cover}(s)$) that matches μ .

The proof of Lemma 6 is obtained by showing that for every *minimal* abstract load vector μ with precision i compatible with $\text{Safe}(s)$ we have that $\mu(j) \leq i \cdot \ell \cdot |S|$ for every $1 \leq j \leq d$ such that $\mu(j) \neq \omega$. Since the minimal elements of $\text{Safe}(s)$ are exactly the minimal abstract vectors of precision d compatible with $\text{Safe}(s)$, we obtain the desired result. The claim is proven by induction on i . In the induction step, we pick a minimal abstract vector μ with precision i compatible with s , and choose a component j such that $\mu(j) = \omega$. Then we show that if we replace $\mu(j)$ with some k whose value is bounded by $(i+1) \cdot \ell \cdot |S|$, we yield a minimal compatible abstract vector with precision $i+1$. The proof of this claim is the very core of the whole argument, and it involves several subtle observations about the structure of minimal abstract load vectors. The details are given in [3].

An important consequence of Lemma 6 is the following:

Corollary 7 (Finite reload property). *If $\alpha \in \text{Safe}(s)$ and $\beta(i) = \min\{\alpha(i), d \cdot \ell \cdot |S|\}$ for every $1 \leq i \leq d$, then $\beta \in \text{Safe}(s)$.*

Due to Corollary 7 for every minimal $\alpha \in Safe(s)$ there is a safe strategy which never reloads any resource to more than $d \cdot \ell \cdot |S|$. Thus, we can significantly improve the bound of Lemma 3(2.).

Corollary 8. *If $Safe(s) \neq \emptyset$, then $(d \cdot \ell \cdot |S|, \dots, d \cdot \ell \cdot |S|) \in Safe(s) \cap Cover(s)$.*

Another consequence of Corollary 7 is that one can reduce the problem of computing the minimal elements of $Safe(s)$ to the problem of determining a winning set in a finite-state 2-player safety game with at most $|S| \cdot d^d \cdot \ell^d \cdot |S|^d + 1$ vertices, which is obtained from C by storing the vector of current resource loads explicitly in the states. Whenever we need to reload some resource, it can be safely reloaded to $d \cdot \ell \cdot |S|$, and we simulate this reload be the corresponding transition. Since the winning set in a safety game with n states and m edges can be computed in time linear in $n + m$ [10], we obtain the following:

Corollary 9. *The sets of all minimal elements of all $Safe(s)$ are computable in time $(d \cdot \ell \cdot |S|)^{O(d)}$.*

The complexity bounds for the algorithmic problems (B) and (C) for $Safe(s)$ are given in our next theorem. The proofs of the presented lower bounds are given in [3].

Theorem 10. *Let $\alpha \in \mathbb{Z}_{>0}^d$ and $s \in S$.*

- *The problem whether $\alpha \in Safe(s)$ is **PSPACE-hard** and solvable in time $|\alpha| \cdot (d \cdot \ell \cdot |S|)^{O(d)}$, where $|\alpha|$ is the encoding size of α .*
- *The problem whether α is a minimal vector of $Safe(s)$ is **PSPACE-hard** and solvable in time $|\alpha| \cdot (d \cdot \ell \cdot |S|)^{O(d)}$, where $|\alpha|$ is the encoding size of α .*
- *The set of all minimal vectors of $Safe(s)$ is computable in time $(d \cdot \ell \cdot |S|)^{O(d)}$.*

Now we provide analogous results for $Cover(s)$. Note that deciding the membership to $Cover(s)$ is trivially reducible to the problem of computing the winning region in a finite-state game obtained from C by constraining the vectors of current resource loads by α . Computing the minimal elements of $Cover(s)$ is more problematic. One is tempted to conclude that all components of the minimal vectors for each $Cover(s)$ are bounded by a “small” number, analogously to Lemma 6. In this case, we obtained only the following bound, which is still polynomial for every fixed d and ℓ , but grows *double-exponentially* in d . The question whether this bound can be lowered is left open, and seems to require a deeper insight into the structure of covering vectors.

Lemma 11. *For every $s \in S$ and every minimal $\alpha \in Cover(s)$ we have that $\alpha(i) \leq (d \cdot \ell \cdot |S|)^{d!}$ for every $1 \leq i \leq d$.*

The proof of Lemma 11 is given in [3]. It is based on re-using and modifying some ideas introduced in [4] for general eVASS games. The following theorem sums up the complexity bounds for problems (B) and (C) for $Cover(s)$.

Theorem 12. *Let $\alpha \in \mathbb{Z}_{>0}^d$ and $s \in S$.*

- *The problem whether $\alpha \in Cover(s)$ is **PSPACE-hard** and solvable in $O(\Lambda^2 \cdot |S|^2)$ time, where $\Lambda = \prod_{i=1}^d \alpha(i)$.*
- *The problem whether α is a minimal element of $Cover(s)$ is **PSPACE-hard** and solvable in $O(d \cdot \Lambda^2 \cdot |S|^2)$ time, where $\Lambda = \prod_{i=1}^d \alpha(i)$.*
- *The set of all minimal vectors of $Cover(s)$ is computable in $(d \cdot \ell \cdot |S|)^{O(d \cdot d!)} \cdot O(\Lambda^2 \cdot |S|^2)$ time.*

4 Algorithms for One-Player and Decreasing Consumption Games

In this section we present more efficient algorithms for the two subclasses of decreasing and one-player consumption games. Observe that these special classes of games can still retain a rich modeling power. In particular, the decreasing subclass is quite natural as systems that do not decrease some of the resources for a long time most probably stopped working completely (also recall that the game considered in Section I is decreasing). One-player consumption games are useful for modeling a large variety of scheduling problems, as it is illustrated in the following example.

Consider the following (a bit idealized) problem of supplying shops with goods such as, e.g., bottles of drinking water. This problem may be described as follows: Imagine a map with c cities connected by roads, n of these cities contain shops to be supplied, k cities contain warehouses with huge amounts of the goods that should be distributed among the shops. The company distributing the goods owns d cars, each car has a bounded capacity. The goal is to distribute the goods from warehouses to all shops in as short time as possible. This situation can be modeled using a one-player consumption game as follows. States would be tuples of the form (c_1, \dots, c_d, A) where each $c_i \in \{1, \dots, c\}$ corresponds to the city in which the i -th car is currently located, $A \subseteq \{1, \dots, n\}$ lists the shops that have already been supplied (initially $A = \emptyset$ and the goal is to reach $A = \{1, \dots, n\}$). Loads of individual cars and the total time would be modelled by a vector of resources, $(\ell(1), \dots, \ell(d), t)$, where each $\ell(i)$ models the current load of the i -th car and t models the amount of time which elapsed from the beginning (this resource is steadily decreased until $A = \{1, \dots, n\}$). Player \square chooses where each car should go next. Whenever the i -th car visits a city with a warehouse, the corresponding resource $\ell(i)$ may be reloaded. Whenever the i -th car visits a city containing a shop, player \square may choose to supply the shop, i.e. decrease the resource $\ell(i)$ of the car by the amount demanded by the shop. Now the last component of a minimal safe configuration indicates how much time is needed to supply all shops. A cover configuration indicates not only how much time is needed but also how large cars are needed to supply all shops. This model can be further extended with an information about the fuel spent by the individual cars, etc.

As in the previous section, we fix a consumption game $C = (S, E, (S_\square, S_\diamond), L)$ of dimension d , and we use ℓ to denote the maximal finite $|\delta(i)|$ such that $1 \leq i \leq d$ and δ is a label of some transition. We first establish the complexity of emptiness and membership problem, and then present an algorithm to compute the minimal safe configurations.

4.1 The Emptiness and Membership Problems

We first establish the complexity of the emptiness problem for decreasing games by a polynomial time reduction to *generalized Büchi games*. A generalized Büchi game is a tuple $\mathcal{B} = (V, \mapsto, (V_\square, V_\diamond), B)$, where $(V, \mapsto, (V_\square, V_\diamond))$ is a 2-player game with finitely many vertices, and $B = \{F_1, \dots, F_m\}$, where $m \geq 1$ and $F_i \subseteq \mapsto$ for all $1 \leq i \leq m$. We say that infinite path w *satisfies* the generalized Büchi condition defined by B iff $\inf(w) \cap F_i \neq \emptyset$ for every $1 \leq i \leq m$. A strategy $\sigma \in \Sigma_{\mathcal{B}}$ is *winning* in $v \in V$ if for every $\pi \in \Pi_{\mathcal{B}}$ we have that the $Play_{\sigma, \pi}(v)$ satisfies the generalized Büchi condition. The

problem whether player \square has a winning strategy in state s can be decided in polynomial time, with an algorithm of complexity $O(|V| \cdot |\mapsto| \cdot m)$ (see [8]).

We claim that the following holds:

Lemma 13. *If C is a decreasing game, then $\text{Safe}(s) \neq \emptyset$ if and only if the player \square has winning strategy in generalized Büchi game $\mathcal{B}_C = (S, E, (S_\square, S_\diamond), \{R_1, \dots, R_d\})$, where for each $1 \leq i \leq d$ we have $R_i = \{(s, t) \in E \mid L(s, t)(i) = \omega\}$.*

Previous lemma immediately gives us that the emptiness of $\text{Safe}(s)$ in decreasing games is decidable in time $O(|S| \cdot |E| \cdot d)$. We now argue that the emptiness of $\text{Safe}(s)$ for one-player games can also be achieved in polynomial time. Note that from Lemma 3 we have that $\text{Safe}(s) \neq \emptyset$ if and only if player \square has a winning strategy in state s of one-player Streett game S_C . The problem of deciding the existence of winning strategy in one-player Streett game is exactly the nonemptiness problem for Streett automata that can be solved in time $O((|S| \cdot d + |E|) \cdot \min\{|S|, d\})$ [11].

Theorem 14. *Given a consumption game C and a state s , the emptiness problems of whether $\text{Safe}(s) = \emptyset$ and $\text{Cover}(s) = \emptyset$ can be decided in time $O(|S| \cdot |E| \cdot d)$ if C is decreasing, and in time $O((|S| \cdot d + |E|) \cdot \min\{|S|, d\})$ if C is a one-player game.*

We now study the complexity of the membership problem for $\text{Safe}(s)$. We prove two key lemmas that bound the number of steps before all resources are reloaded. The key idea is to make player \square reload resources as soon as possible. Formally, we say that a play $\text{Play}_{\sigma, \pi}(s, \alpha)$ induced by a sequence of transitions $s_0 \xrightarrow{\delta_1} \dots \xrightarrow{\delta_k} s_k$ reloads i -th resource in j -th step if $\delta_j(i) = \omega$. We first present a lemma for decreasing games and then for one-player games.

Lemma 15. *Consider a decreasing consumption game C and a configuration (s, α) such that $\alpha \in \text{Safe}(s)$. There is a safe strategy σ for player \square in (s, α) such that every $\text{Play}_{\sigma, \pi}(s, \alpha)$ reloads all resources in the first $d \cdot |S|$ steps.*

Now let us consider one-player games. As player \diamond has only one trivial strategy, π , we write only $\text{Play}_\sigma(s, \alpha)$ instead of $\text{Play}_{\sigma, \pi}(s, \alpha)$.

Lemma 16. *Consider a one-player consumption game C and a configuration (s, α) such that $\alpha \in \text{Safe}(s)$. There is a safe strategy σ for player \square in (s, α) such that for the $\text{Play}_\sigma(s, \alpha)$ and every $1 \leq i \leq d$ we have that either the i -th resource is reloaded in the first $d \cdot |S|$ steps, or it is never decreased from the $(d \cdot |S| + 1)$ -st step on.*

As a consequence of Lemma 15, Lemma 16 and the hardness results presented in [3] we obtain the following:

Theorem 17. *The membership problem of whether $\alpha \in \text{Safe}(s)$ is **NP**-complete for one-player consumption games and **PSPACE**-complete for decreasing consumption games. The problem whether α is a minimal element of $\text{Safe}(s)$ is **DP**-complete for one-player consumption games and **PSPACE**-complete for decreasing consumption games.*

4.2 Minimal Safe Configurations and Multi-distance Reachability

In the rest of the paper we present algorithms for computing the minimal safe configurations in one-player and decreasing consumption games. Both algorithms use the iterative algorithm for *multi-distance reachability problem*, which is described below, as a subprocedure. Although their worst-case complexity is the same as the complexity of generic algorithm from Section 3, we still deem them to be more suitable for practical computation due to some of their properties that we state here in advance:

- The generic algorithm always constructs game of size $(|S| \cdot d \cdot \ell)^{O(d)}$. In contrast, algorithms based on solving multi-distance reachability construct a game whose size is linear in size of C for every fixed choice of parameter d .
- The multi-distance reachability algorithms iteratively construct sets of configurations that are safe but may not be minimal before the algorithm stops. Although the time complexity of this iterative computation is $(|S| \cdot d \cdot \ell)^{O(d)}$ at worst, it may be the case that the computation terminates much earlier. Thus, these algorithms have a chance to terminate earlier than in $(|S| \cdot d \cdot \ell)^{O(d)}$ steps (unlike the generic algorithm, where the necessary construction of the “large” safety game always requires this number of steps).
- Moreover, the algorithm for one-player games presented in Section 4.3 decomposes the problem into many parallel subtasks that can be processed independently.

Let \mathcal{D} denote a d -dimensional consumption game with transitions labeled by vectors over $\mathbb{Z}_{\leq 0}$ (i.e. there is no ω in any label). Also denote D the set of states of game \mathcal{D} . We say that vector α is a *safe multi-distance* (or just *safe distance*) from state s to state r if there is a strategy σ for player \square such that for any strategy π for player \diamond the infinite path $Play_{\sigma,\pi}(s, \alpha)$ visits a configuration of the form (r, β) . That is, α is a safe distance from s to r if player \square can enforce reaching r from s in such a way that the total decrease in resource values is less than α .

We denote by $Safe_{\mathcal{D}}(s, r)$ the set of all safe distances from s to r in \mathcal{D} , and by $\lambda_{\mathcal{D}}(s, r)$ the set of all minimal elements of $Safe_{\mathcal{D}}(s, r)$. If $Safe_{\mathcal{D}}(s, r) = \emptyset$, then we set $\lambda_{\mathcal{D}}(s, r) = \{(\infty, \dots, \infty)\}$, where the symbol ∞ is treated accordingly with the usual conventions (for any $c \in \mathbb{Z}$ we have $\infty - c = \infty$, $c < \infty$; we do not use the ω symbol to avoid confusions).

We present a simple fixed-point iterative algorithm which computes the set of minimal safe distances from s to r . Apart from the standard set operations, the algorithm uses the following operations on sets of vectors: for a given set M and a given vector α , the operation $\text{min-set}(M)$ returns the set of minimal elements of M , and $M - \alpha$ returns the set $\{\beta - \alpha \mid \beta \in M\}$. Further, given a sequence of sets of vectors M_1, \dots, M_m the operation $\text{cwm}(M_1, \dots, M_m)$ returns the set $\{\alpha_1 \vee \dots \vee \alpha_m \mid \alpha_1 \in M_1, \dots, \alpha_m \in M_m\}$, where each $\alpha_1 \vee \dots \vee \alpha_m$ denotes a component-wise maximum of the vectors $\alpha_1, \dots, \alpha_m$.

Technically, the algorithm iteratively solves the following optimality equations: for any state q with outgoing transitions $q \xrightarrow{\delta_1} q_1, \dots, q \xrightarrow{\delta_m} q_m$ we have that

$$\lambda_{\mathcal{D}}(q, r) = \begin{cases} \text{min-set}(\lambda_{\mathcal{D}}(q_1, r) - \delta_1 \cup \dots \cup \lambda_{\mathcal{D}}(q_m, r) - \delta_m) & \text{if } q \in D_{\square} \\ \text{min-set}(\text{cwm}(\lambda_{\mathcal{D}}(q_1, r) - \delta_1, \dots, \lambda_{\mathcal{D}}(q_m, r) - \delta_m)) & \text{if } q \in D_{\diamond} \end{cases}$$

The algorithm iteratively computes the k -step approximations of $\lambda_{\mathcal{D}}(q, r)$, which are denoted by $\lambda_{\mathcal{D}}^k(q, r)$. Intuitively, each set $\lambda_{\mathcal{D}}^k(q, r)$ consists of all minimal safe

distances from q to r over all plays with at most k steps. The set $\lambda_{\mathcal{D}}^0(q, r)$ is initialized to $\{\infty, \dots, \infty\}$ for $q \neq r$, and to $\{(1, \dots, 1)\}$ for $q = r$. Each $\lambda_{\mathcal{D}}^{k+1}(q, r)$ is computed from $\lambda_{\mathcal{D}}^k(q, r)$ using the above optimality equations until a fixed point is reached. In [3] we show that this fixed point is the correct solution for the minimal multi-distance problem.

Since the algorithm is based on standard methods, we omit its presentation (which can be found in [3]) and state only the final result. We call *branching degree* of \mathcal{D} the maximal number of transitions outgoing from any state of \mathcal{D} .

Theorem 18. *There is an iterative procedure $\text{Min-dist}(\mathcal{D}, s, r)$ that correctly computes the set of minimal safe distances from s to r in time $O(|\mathcal{D}| \cdot a \cdot b \cdot N^2)$, where b is the branching degree of \mathcal{D} , a is the length of a longest acyclic path in \mathcal{D} and $N = \max_{0 \leq k \leq a} |\lambda_{\mathcal{D}}^k(q, r)|$.*

Moreover, the procedure requires at most a iterations to converge to the correct solution and thus the resulting set $\lambda_{\mathcal{D}}(s, r)$ has size at most N . Finally, the number N can be bounded from above by $(a \cdot \ell)^d$.

Note that the complexity of the procedure $\text{Min-dist}(\mathcal{D}, s, r)$ crucially depends on parameter N . The bound on N presented in the previous theorem follows from the obvious fact that components of all vectors in $\lambda_{\mathcal{D}}^k(s, r)$ are either all equal to ∞ or are all bounded from above by $k \cdot \ell$. However, for concrete instances the value of N can be substantially smaller. For example, if the consumption game \mathcal{D} models some real-world problem, then it can be expected that the number of k -step minimal distances from states of \mathcal{D} to r is small, because changes in resources are not entirely independent in these models (e.g., action that consumes a large amount of some resource may consume a large amount of some other resources as well). This observation forms the core of our claim that algorithms based on multi-distance reachability may terminate much earlier than the generic algorithm from Section 3.

4.3 Computing $\text{Safe}(s)$ in One-Player Consumption Games

Now we present an algorithm for computing minimal elements of $\text{Safe}(s)$ in one-player consumption games. The algorithm computes the solution by solving several instances of minimum multi-distance reachability problem. We assume that all states s with $\text{Safe}(s) = \emptyset$ were removed from the game. This can be done in polynomial time using the algorithm for emptiness (see Theorem 14).

We denote by $\Pi(d)$ the set of all permutations of the set $\{1, \dots, d\}$. We view each element of $\Pi(d)$ as a finite sequence $\pi_1 \dots \pi_d$, e.g., $\Pi(2) = \{12, 21\}$. We use the standard notation π for permutations: confusion with strategies of player \diamond should not arise since $S_{\diamond} = \emptyset$ in one-player games.

We say that a play $\text{Play}_{\sigma}(s, \alpha)$ matches a permutation π if for every $1 \leq i < j \leq d$ the following holds: If the π_j -th resource is reloaded along $\text{Play}_{\sigma}(s, \alpha)$, then the π_i -th resource is also reloaded along this play and the first reload of π_i -th resource occurs before or at the same time as the first reload of π_j -th resource. A configuration (s, α) matches π if there is a strategy σ that is safe in (s, α) and $\text{Play}_{\sigma}(s, \alpha)$ matches π . We denote by $\text{Safe}(s, \pi)$ the set of all vectors α such that (s, α) matches π . Note that $\text{Safe}(s) = \bigcup_{\pi \in \Pi(d)} \text{Safe}(s, \pi)$.

As indicated by the above equality, computation of safe configurations in C reduces to the problem of computing, for every permutation π , safe configurations that match π . The latter problem, in turn, easily reduces to the problem of computing safe multi-distances in specific one-player consumption games $C(\pi)$. Intuitively, each game $C(\pi)$ simulates the game C where the resources are forced to be reloaded in the order specified by π . So the states of each $C(\pi)$ are pairs (s, k) where s corresponds to the current state of the original game and k indicates that the first k resources, in the permutation π , have already been reloaded. Now the crucial point is that if the first k resources have been reloaded when some configuration $c = (s, \beta)$ of the original game is visited, and there is a safe strategy in c which does not decrease any of the resources with the index greater than k , then we may safely conclude that the *initial* configuration is safe. So, in such a case we put a transition from the state (s, k) of $C(\pi)$ to a distinguished target state r (whether or not to put in such a transition can be decided in polynomial time due to Theorem 14). Other transitions of $C(\pi)$ correspond to transitions of C except that they have to update the information about already reloaded resources, cannot skip any resource in the permutation (such transitions are removed), and the components indexed by π_1, \dots, π_k are substituted with 0 in transitions incoming to states of the form (q, k) (since already reloaded resources become unimportant as indicated by the above observation).

A complete construction of $C(\pi)$ is presented in [3] as a part of a formal proof of the following theorem:

Theorem 19. *For every permutation π there is a polynomial time constructible consumption game $C(\pi)$ of size $O(|S| \cdot d)$ and branching degree $O(|S|)$ such that for every vector α we have that $\alpha \in Safe(s, \pi)$ in C iff α is a safe distance from $(s, 0)$ to r in $C(\pi)$.*

By the previous theorem, every minimal element of $Safe(s)$ is an element of $\lambda_{C(\pi)}((s, 0), r)$ for at least one permutation π . Our algorithm examines all permutations $\pi \in \Pi(d)$, and for every permutation it constructs game $C(\pi)$ and computes $\lambda_{C(\pi)}((s, 0), r)$ using the procedure Min-dist from Theorem 18. The algorithm also stores the set of all minimal vectors that appear in some $\lambda_{C(\pi)}((s, 0), r)$. In this way, the algorithm eventually finds all minimal elements of $Safe(s)$. The pseudocode of the algorithm is presented in [3].

From complexity bounds of Theorems 14 and 18 we obtain that the worst case running time of this algorithm is $d! \cdot (|S| \cdot \ell \cdot d)^{O(d)}$. In contrast with the generic algorithm of Section 3 that constructs an exponentially large safety game, the algorithm of this section computes $d!$ “small” instances of the minimal multi-distance reachability problem. We can solve many of these instances in parallel. Moreover, as argued in previous section, each call of $Min-dist(C(\pi), (s, 0), r)$ may have much better running time than the worst-case upper bound suggests.

4.4 Computing $Safe(s)$ in Decreasing Consumption Games

We now turn our attention to computing minimal elements of $Safe(s)$ in decreasing games. The main idea is again to reduce this task to the computation of minimal multi-distances in certain consumption game. We again assume that states with $Safe(s) = \emptyset$ were removed from the game.

The core of the reduction is the following observation: if C is decreasing, then $\alpha \in Safe(s)$ iff player \square is able to ensure that the play satisfies these two conditions: all resources are reloaded somewhere along the play; and the i -th resource is reloaded for the first time before it is decreased by at least $\alpha(i)$, for every $1 \leq i \leq d$. Now if we augment the states of C with an information about which resources have been reloaded at least once in previous steps, then the objective of player \square is actually to reach a state which tells us that all resources were reloaded at least once.

So the algorithm constructs a game \widehat{C} by augmenting states of C with an information about which resources have been reloaded at least once, and by substituting updates of already reloaded resources (i.e., the corresponding components of the labels) with zeros. Note that the construction of \widehat{C} closely resembles the construction of games $C(\pi)$ from the previous section. However, in two-player case we cannot fix an order in which resources are to be reloaded, because the optimal order depends on a strategy chosen by player \diamond . Thus, we need to remember exactly which resources have been reloaded in the past (we only need to remember the set of resources that have been reloaded, but not the order in which they were reloaded).

The formal construction of \widehat{C} can be found in [3] along with a proof of the following theorem.

Theorem 20. *There is a consumption game \widehat{C} of size $O(2^d \cdot |S|)$, branching degree $O(S)$ and with maximal acyclic path of length $O(|S| \cdot d)$, with the following properties: \widehat{C} is constructible in time $O(2^d \cdot (|S| + |E|))$ and for every vector α we have $\alpha \in Safe(s)$ in C iff α is a safe distance from (s, \emptyset) to r in \widehat{C} .*

The previous theorem shows that we can find minimal elements of $Safe(s)$ with a single call of procedure $Min\text{-}dist(\widehat{C}, (s, \emptyset), r)$. Straightforward complexity analysis reveals that the worst-case running time of this algorithm is $(|S| \cdot d \cdot \ell)^{O(d)}$. However, the game \widehat{C} constructed during the computation is still smaller than the safety game constructed by the generic algorithm of Section 3. Moreover, the length of the longest acyclic path in \widehat{C} is bounded by $|S| \cdot d$, so the procedure $Min\text{-}dist$ does not have to perform many iterations, despite the exponential size of \widehat{C} . Finally, let us once again recall that the procedure $Min\text{-}dist(\widehat{C}, (s, \emptyset), r)$ may actually require much less than $(|S| \cdot d \cdot \ell)^{O(d)}$ steps.

5 Conclusions

As it is witnessed by the results presented in previous sections, consumption games represent a convenient trade-off between expressive power and computational tractability. The presented theory obviously needs further development before it is implemented in working software tools. Some of the issues are not yet fully understood, and there are also other well-motivated problems about consumption games which were not considered in this paper. The list of important open problems includes the following:

- Improve the complexity of algorithms for $Cover(s)$. This requires further insights into the structure of these sets.
- Find efficient controller synthesis algorithms for objectives that combine safety with other linear-time properties. That is, decide whether player \square has a safe strategy such that a play satisfies a given LTL property no matter what player \diamond does.

- Find algorithms for more complicated optimization problems, where the individual resources may have different priorities. For example, it may happen that fuel consumption or the price of batteries with large capacity are much more important than the time spent, and in that case we might want to optimize some weight function over the tuple of all resources. It may happen (and we have concrete examples) that some of these problems are actually solvable even more efficiently than the general ones where all resources are treated equally w.r.t. their importance.

The above list is surely incomplete. The problem of optimal resource consumption is rather generic and appears in many different contexts, which may generate other interesting questions about consumption games.

References

1. Beeri, C.: On the membership problem for functional and multivalued dependencies in relational databases. *ACM Trans. on Database Systems* 5, 241–259 (1980)
2. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints. In: Johansson, K., Yi, W. (eds.) HSCC 2010, pp. 61–70. ACM (2010)
3. Brázdil, T., Chatterjee, K., Kučera, A., Novotný, P.: Efficient controller synthesis for consumption games with multiple resource types, CoRR abs/1202.0796 (2012)
4. Brázdil, T., Jančar, P., Kučera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010)
5. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource Interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
6. Chaloupka, J.: Z-Reachability Problem for Games on 2-Dimensional Vector Addition Systems with States Is in P. In: Kučera, A., Potapov, I. (eds.) RP 2010. LNCS, vol. 6227, pp. 104–119. Springer, Heidelberg (2010)
7. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: Lodaya, K., Mahajan, M. (eds.) Proc. of FSTTCS 2010. LIPIcs, vol. 8, pp. 505–516. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
8. Dziembowski, S., Jurdzinski, M., Walukiewicz, I.: How much memory is needed to win infinite games? In: LICS 1997, pp. 99–110. IEEE (1997)
9. Emerson, E., Jutla, C.: The complexity of tree automata and logics of programs. In: FOCS 1988, pp. 328–337. IEEE (1988)
10. Immerman, N.: Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences* 22(3), 384–406 (1981)
11. Kurshan, R.: Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton University Press (1994)
12. Lipton, R.: The reachability problem requires exponential space. Technical report 62, Yale University (1976)
13. Piterman, N., Pnueli, A.: Faster solution of Rabin and Streett games. In: LICS 2006, pp. 275–284. IEEE (2006)

ACTL \cap LTL Synthesis*

Rüdiger Ehlers

Reactive Systems Group, Saarland University

Abstract. We study the synthesis problem for specifications of the common fragment of ACTL (computation tree logic with only universal path quantification) and LTL (linear-time temporal logic). Key to this setting is a novel construction for translating properties from LTL to very-weak automata, whenever possible. Such automata are structurally simple and thus amenable to optimizations as well as symbolic implementations.

Based on this novel construction, we describe a synthesis approach that inherits the efficiency of generalized reactivity(1) synthesis [27], but is significantly richer in terms of expressivity.

1 Introduction

Synthesizing reactive systems from functional specifications is an ambitious challenge. It combines the correctness assurance that systems obtain after model checking with the advantage to skip the manual construction step for the desired system. As a consequence, a rich line of research has emerged, witnessed by the fact that recently, off-the-shelf tools for this task have become available.

A central question in synthesis is: *what is the right specification language that allows us to tackle the synthesis problem for its members efficiently, while still having enough expressivity to capture the specifications that system designers want to write?*

Some recent approaches focused on supporting full linear-time temporal logic as the specification language. While the synthesis problem for such specifications was shown to be 2EXPTIME-complete, by focusing on specifications of the form that engineers tend to write, significant progress could recently be obtained for full LTL [17][13]. Still, it is not hard to write small specifications that cannot be tackled by such tools.

At the same time, there are numerous techniques that trade the high expressivity of logics such as LTL against the computational advantages of only having to deal with structurally simpler specifications. A prominent approach of this kind is *generalized reactivity(1) synthesis* [27]. It targets specifications that consist of some set of assumptions (which we can assume the environment of the system to fulfill) and some set of guarantees that the system needs to fulfill. Both assumptions and guarantees can contain only safety properties that relate the input and output in one computation cycle with the input and output in the next computation cycle and basic liveness properties over current input and output. In order to encode more complex properties, the output of the system to be designed can be widened and the additional bits can be used to stitch together

* This work was supported by the DFG as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS).

more complex properties. Getting such an encoding right and efficient is manual and cumbersome work, which is why Somenzi and Sohail coined the term “*pre-synthesis*” for such an operation [28][1].

It is apparent that there is a desperate need for a sweet spot between the high expressivity but low performance that full LTL synthesis approaches offer, and fast but low-level synthesis approaches such as generalized reactivity(1) synthesis, where currently, pre-synthesis is crucial to its performance.

In this paper, we present $ACTL \cap LTL$ synthesis as a solution to this problem. Our approach targets specifications of the form $\bigwedge_{a \in \text{Assumptions}} a \rightarrow \bigwedge_{g \in \text{Guarantees}} g$, where all assumptions and guarantees are written in LTL, with the restriction that they must also be representable in ACTL, i.e., computation tree logic with only universal path quantification. We reduce the synthesis problem for such specifications to solving symbolically represented three-color parity games, which is the reasoning framework from which also generalized reactivity(1) synthesis takes its good efficiency. In particular, such games can be solved in time quadratic in the number of positions (see, e.g., [1]).

The reason why $ACTL \cap LTL$ is such an interesting fragment for synthesis is the fact that the fragment has *universal very-weak automata* as the characterizing automaton class. These automata do not only allow the application of simple, yet effective minimization algorithms, but give rise to a straight-forward efficient symbolic encoding into binary decision diagrams (BDDs), without the need for pre-synthesis. Alternatively, other symbolic data structures such as *anti-chains* [16] can also be used, but for the simplicity of the initial evaluation of the approach in this paper, we use BDDs.

For best performance in solving the parity games that we build in our approach, we present a novel construction that defers choosing the assumption and guarantee parts to be satisfied next to the system player and the environment player, respectively. This keeps the number of iterations that need to be performed in the fixed-point based game solving process small and leads to short computation times of the game solving process.

The contribution of this paper is threefold. First of all, it describes a new efficient synthesis workflow for the common fragment of ACTL and LTL. Secondly, it describes the first algorithm for translating an LTL formula that lies in this common fragment into its characterizing automaton class, i.e., universal very-weak automata. As a corollary, we obtain a translation algorithm from LTL to ACTL, whenever possible. Third, we introduce a technique to speed up the game solving process for generalized reactivity(1) games by letting the two players in the game choose the next obligation for the respective other player instead of using counters as in previous approaches.

We start with preliminaries in Sect. 2, where we discuss the basic properties of very-weak automata. Then, we describe the construction to obtain universal very-weak automata from LTL formulas that are also representable in ACTL. Afterwards, we present the smart reduction of our synthesis problem to three-color parity games in Sect. 4. Section 5 then discusses the twists and tricks for solving parity games symbolically in an efficient way and describes how a winning strategy that represents an implementation satisfying the specification can be extracted. Finally, Sect. 6 contains an experimental evaluation of the approach using a prototype toolset for the overall workflow. We conclude in Sect. 7.

2 Preliminaries

Basics: Given a (finite) alphabet Σ , we denote the sets of finite and infinite words of Σ as Σ^* and Σ^ω , respectively. Sets of words are called *languages*. A useful tool for representing languages over finite words are regular expressions, and ω -regular expressions are regular expressions that are enriched by the $(\cdot)^\omega$ operator, which denotes infinite repetition. This way, languages over infinite words can be expressed.

Given some monotone function $f : 2^X \rightarrow 2^X$ for some finite set X , we define $\mu^0.f = \emptyset$, $\nu^0.f = X$ and for every $i > 0$, set $\mu^i.f = (f \circ \mu^{i-1}.f)$ and $\nu^i.f = (f \circ \nu^{i-1}.f)$. For a monotone function f and finite X , it is assured that the series $\mu^0.f, \mu^1.f, \mu^2.f \dots$ and $\nu^0.f, \nu^1.f, \nu^2.f \dots$ converge to some limit functions, which we denote by $\mu.f$ and $\nu.f$, respectively.

Automata: For reasoning about (ω -)regular languages, *automata* are a suitable tool. In this paper, we will be concerned with *deterministic*, *non-deterministic*, *non-deterministic very-weak* and *universal very-weak* automata over finite and infinite words. For all of these types, the automata are described by tuples $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ with the set of states Q , the alphabet Σ , the set of initial states $Q_0 \subseteq Q$, and the transition function $\delta : Q \times \Sigma \rightarrow 2^Q$. For non-deterministic or deterministic automata, $F \subseteq Q$ is called the set of *accepting states*, whereas for universal automata, $F \subseteq Q$ denotes the set of *rejecting states*. For deterministic automata, we require that $|Q_0| = 1$ and that for every $(q, x) \in Q \times \Sigma$, we have $|\delta(q, x)| \leq 1$. For very-weak automata, we require them to have an order $f : Q \rightarrow \mathbb{N}$ on the states such that for every transition from a state q to a state q' for some $x \in \Sigma$ (i.e., $q' \in \delta(q, x)$), if $q' \neq q$, then $f(q') > f(q)$. Figure 1 contains examples of very-weak automata. Intuitively, the order requires the automaton to be representable in a figure such that all non-self-loop transitions lead from top to bottom.

Given a word $w = w_0w_1w_2 \dots w_n \in \Sigma^*$, we say that $\pi = \pi_0\pi_1 \dots \pi_{n+1}$ is a finite run for \mathcal{A} and w if $\pi_0 \in Q_0$ and for $0 \leq i \leq n$, $\pi_{i+1} \in \delta(\pi_i, w_i)$. Likewise, for a word $w = w_0w_1w_2 \dots \in \Sigma^\omega$, we say that $\pi = \pi_0\pi_1 \dots$ is an infinite run for \mathcal{A} and w if $\pi_0 \in Q_0$ and for all $i \in \mathbb{N}$, $\pi_{i+1} \in \delta(\pi_i, w_i)$.

A non-deterministic (NFA), non-deterministic very-weak (NVWF) or deterministic (DFA) automaton over finite words accepts all finite words that have some run that ends in an accepting state. A universal automaton over finite words accepts all finite words for which all runs do not end in a rejecting state. A non-deterministic automaton over infinite words accepts all infinite words that have some run that visits accepting states infinitely often. A universal very-weak automaton over infinite words (UVW) accepts all infinite words for which all runs visit rejecting states only finitely often.

We say that two automata are equivalent if they accept the same set of words. This set of words is also called their *language*. We define the *language of a state* q to mean the language of the automaton that results from setting the initial states to $\{q\}$. The functions $\hat{\delta} : 2^Q \times 2^X \rightarrow 2^Q$ and $\hat{\delta}^* : 2^Q \times 2^X \rightarrow 2^Q$ with $\hat{\delta}(Q', X) = \bigcup_{\{q' \in Q', x \in X\}} \delta(q', x)$ and $\hat{\delta}^*(Q', X) = \{q' \in Q \mid \exists k \in \mathbb{N}, x_1, x_2, \dots, x_k \in X, q_1, q_2, \dots, q_{k+1} \in Q. (q_1 \in Q' \wedge q_k = q' \wedge \forall 1 \leq i \leq k. q_{i+1} \in \delta(q_i, x_i))\}$ will simplify the presentation in Sect. 3. Deterministic automata over finite words also appear as *distance automata* in this paper. The only difference to non-distance automata is the

fact that for these, we have $\delta : Q \times \Sigma \rightarrow 2^{Q \times \{0,1\}}$. We assign with each of their runs the *accumulated cost*, obtained by adding all of the second components of the transition target tuples for the transitions along the run. The cost of a word is the minimal cost of an accepting run.

Labeled parity games: A parity game is defined as a tuple $\mathcal{G} = (V_0, V_1, \Sigma_0, \Sigma_1, E_0, E_1, v_0, c)$ with the game position sets V_0 and V_1 for player 0 and player 1, respectively, the action sets Σ_0 and Σ_1 , the edge functions $E_0 : V_0 \times \Sigma_0 \rightarrow V_1$ and $E_1 : V_1 \times \Sigma_1 \rightarrow V_0$, the initial position $v_0 \in V_0$, and the coloring function $c : (V_0 \uplus V_1) \rightarrow \mathbb{N}$.

A decision sequence in \mathcal{G} is a sequence $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \dots$ such that for all $i \in \mathbb{N}$, $\rho_i^0 \in \Sigma_0$ and $\rho_i^1 \in \Sigma_1$. A decision sequence ρ induces an infinite play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \dots$ if $\pi_0^0 = v_0$ and for all $i \in \mathbb{N}$ and $p \in \{0, 1\}$, $E_p(\pi_i^p, \rho_i^p) = \pi_{i+p}^{1-p}$.

Given a play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \dots$, we say that π is winning for player 1 if $\max\{c(v) \mid v \in V_0 \uplus V_1, v \in \inf(\pi)\}$ is even for the function \inf mapping a sequence onto the set of elements that appear infinitely often in the sequence. If a play is not winning for player 1, it is winning for player 0.

Given some parity game $\mathcal{G} = (V_0, V_1, \Sigma_0, \Sigma_1, E_0, E_1, v_0, c)$, a strategy for player 0 is a function $f_0 : (\Sigma_0 \times \Sigma_1)^* \rightarrow \Sigma_0$. Likewise, a strategy for player 1 is a function $f_1 : (\Sigma_0 \times \Sigma_1)^* \times \Sigma_0 \rightarrow \Sigma_1$. In both cases, a strategy maps prefix decision sequences to an action to be chosen next. A decision sequence $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \dots$ is said to be in correspondence to f_p for some $p \in \{0, 1\}$ if for every $i \in \mathbb{N}$, we have $\rho_i^p = f_p(\rho_0^0 \rho_0^1 \dots \rho_{i+p-1}^{1-p})$. A strategy is winning for player p if all plays in the game that are induced by some decision sequence that is in correspondence to f_p are winning for player p . It is a well-known fact that for parity games, there exists a winning strategy for precisely one of the players (see, e.g., [26][22]).

Labeled parity games for synthesis: Parity games are a computation model for systems that interact with their environment. For the scope of this paper, let us assume that player 0 represents the *environment* of a system that we want to synthesize, and player 1 represents the system itself. The action set of player 0 corresponds to the inputs to the system and the action set of player 1 corresponds to the output. Given a language L over infinite words for the desired properties of a system, the main idea when building a parity game for synthesis is to ensure that the decision sequences that induce winning plays are the ones that, when read as words, are in L . If the game is then found to be winning for the system player, we can take a *strategy* for that player to win the game and read it as a *Mealy* automaton that is guaranteed to satisfy the specification. Note that all constructions in this paper can equally be used for a *Moore* automaton computation model. The two players then swap roles in this case.

Linear-time temporal logic: Linear-time temporal logic (LTL) is a popular formalism to describe properties of systems to be synthesized or verified. LTL formulas are built inductively from atomic propositions in some set AP and sub-formulas using the Boolean operators \neg , \vee , \wedge , and the temporal operators X, F, G, and U. Given an infinite word $w = w_0 w_1 w_2 \in (2^{\text{AP}})^\omega$, a LTL formula over AP either holds on w or not. The words for which an LTL formula holds are also called its *models*. A full definition of LTL can be found in [12][16][18].

Properties of Very-Weak Automata

As foundation for the constructions of the sections to come, we discuss some properties of very-weak automata over finite and infinite words here. Given two automata, we call computing a third automaton that represents the set of words that are accepted by both automata *taking their conjunction*, while *taking their disjunction* refers to computing a third automaton that accepts all words that are accepted by either of the two input automata.

Proposition 1. *Universal and non-deterministic very-weak automata over infinite and finite words are closed under disjunction and conjunction. Given two very-weak automata \mathcal{A} and \mathcal{A}' with state sets Q and Q' , we can compute their disjunctions and conjunctions in polynomial time, with the following state counts of the results:*

1. *for universal automata and taking the conjunction: $|Q| + |Q'|$ states,*
2. *for non-deterministic automata and taking the disjunction: $|Q| + |Q'|$ states,*
3. *for universal automata and taking the disjunction: $|Q| \cdot |Q'|$ states, and*
4. *for non-deterministic automata and taking the conjunction: $|Q| \cdot |Q'|$ states.*

Proof. For the first two cases, the task can be accomplished by just merging the state sets and transitions. For cases 3 and 4, a standard product construction can be applied, with defining those states in the product as rejecting/accepting for which both corresponding states in the factor automata are rejecting/accepting, respectively [20]. \square

Proposition 2. *Every very-weak automaton has an equivalent one of the same type for which no accepting/rejecting state has a non-self-loop outgoing edge (called the separated form of the automaton henceforth).*

Proof. Duplicate every accepting/rejecting state in the automaton and let the duplicate have the same incoming edges. Then, mark the original copy of the state as non-accepting/non-rejecting. The left part of Fig. 11 shows an example of such a state duplication. \square

The fact that every automaton has a separated form allows us to decompose it into a set of so-called *simple chains*:

Definition 1. *Given an alphabet Σ , we call a subset Q' of states of an automaton over Σ a simple chain if there exists a transition order on Q' , i.e., a bijective function $f : Q' \rightarrow \{1, \dots, |Q'|\}$ such that:*

- *only the state q with $f(q) = 1$ is initial,*
- *only the state q with $f(q) = |Q'|$ is accepting/rejecting,*
- *there is no transition in the automaton between a state in Q' and a state not in Q' ,*
- *for every transition from q to q' in the automaton, $f(q) \leq f(q') \leq f(q) + 1$.*

Furthermore, regular expressions that are an unnested concatenation of elements of the form A , A^* , and A^ω for $A \subseteq \Sigma$ are called vermicelli.

As an example, the right-most sequence of states in Fig. 11 is a simple chain and can equivalently be represented as the vermicelli $\Sigma^* a(\bar{b})^* b(\bar{c})^\omega$. Note that every vermicelli can be translated to a language-equivalent set of simple chains.

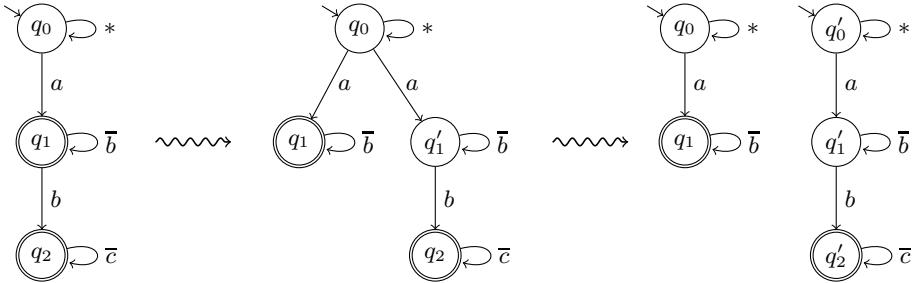


Fig. 1. Example for converting a UVW into separated form and subsequently decomposing it into simple chains. The automata in this example are equivalent to the LTL formula $G(a \rightarrow XF(b \wedge XF\bar{c}))$. We use Boolean combinations of atomic propositions and their negation as edge labels here. For example, \bar{b} refers to all elements $x \in \Sigma = 2^{\text{AP}}$ for which $b \notin x$. Rejecting states are doubly-circled.

Proposition 3. *Every very-weak automaton can be translated to a form in which it only consists of simple chains.*

Proof. Convert the very-weak automaton into separated form and enumerate all paths to leaf nodes along with the self-loops that might possibly be taken. For every of these paths, construct a simple chain. \square

3 Translating LTL Formulas into UVWs

Universal very-weak automata (UVW) were identified as a characterizing automaton class for the intersection of ACTL and LTL by Maidl [25]. She also described an algorithm to check for a given ACTL formula if it lies in the intersection. For the LTL case, Maidl defined a syntactic fragment of it, named LTL^{det}, whose expressivity coincides with that of $\text{ACTL} \cap \text{LTL}$. However, she did not show how to translate an LTL formula into this fragment whenever possible, and the fragment itself is cumbersome to use, as it essentially requires the specifier to describe the structure of a UVW in LTL, and is not even closed under disjunction, although UVW are. Thus, for all practical means, the question how to check for a given LTL formula if it is contained in $\text{ACTL} \cap \text{LTL}$ remained open.

When synthesizing a system, the designer of the system specifies the desired sequence of events, for which linear-time logics are more intuitive to use than branching-time logics. Thus, to use the advantage of universal very-weak automata in actual synthesis tool-chains, the ability to translate from LTL to UVW is highly desirable.

Recently, Bojańczyk [4] gave an algorithm for testing the membership of the set of models of an LTL formula in $\text{ACTL} \cap \text{LTL}$ after the LTL formula has been translated to a deterministic parity automaton. However, the algorithm cannot generate a universal very-weak automaton (UVW) from the parity automaton in case of a positive answer. The reason is that the algorithm is based on searching for so-called *bad patterns* in the automaton. If none of these are present, the deterministic parity automaton is found

to be convertible, but we do not obtain any information about how a UVW for the property might look like. Here, we reduce the problem of constructing a UVW for a given ω -regular language to a sequence of problems over automata for finite words. We modify a procedure by Hashiguchi [19] that builds a distance automaton to check if a given language over finite words can be decomposed into a set of vermicelli (see Def. I). Our modification adds a component to keep track of vermicelli already found. This way, by iteratively searching for vermicellis of increasing length in the language, we eventually find them all and obtain a full language decomposition.

Since Maidl [25, Lemma 11] described a procedure to translate a UVW to an equivalent ACTL formula, we obtain as a corollary also a procedure to translate from LTL to ACTL, whenever possible.

3.1 The Case of Automata over Infinite Words

We have seen that every UVW can be translated to a separated UVW. In a separated UVW, we can distinguish rejecting states by the set of alphabet symbols for which the states have self-loops. If two rejecting states have the same set, we can merge them without changing the language of the automaton. As a corollary, we obtain that a UVW can always be modified such that it is in separated form and has at most $2^{|\Sigma|}$ rejecting states. We will see in this section that obtaining a UVW for a given language L over some alphabet Σ can be done by finding a suitable *decomposition* of the set of words that are not in L among these up to $2^{|\Sigma|}$ rejecting states, and then constructing the rest of the UVW such that words that are mapped to some rejecting state in the decomposition induce runs that eventually enter that rejecting state and stay there forever.

Definition 2. Given a language L over infinite words from the alphabet Σ , we call a function $f : 2^\Sigma \rightarrow 2^{\Sigma^*}$ an end-component decomposition of L if $L = \Sigma^\omega \setminus \bigcup_{X \subseteq \Sigma} (f(X) \cdot X^\omega)$. We call f a maximal end-component decomposition of L if for every $X \subseteq \Sigma$, $f(X) = \{w \in \Sigma^* \mid w \cdot X^\omega \cap L = \emptyset\}$.

Definition 3. Given a separated UVW $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ and an end-component decomposition f , we say that f corresponds to \mathcal{A} if for $(q_1, X_1), \dots, (q_m, X_m)$ being the rejecting states and alphabet symbols under which they have self-loops, we have:

- for all $i \neq j$, $X_i \neq X_j$;
- for all $1 \leq i \leq m$: $f(X_i) = \{w_0 w_1 \dots w_k \in \Sigma^* \mid q_i \in \hat{\delta}(\dots \hat{\delta}(\hat{\delta}(Q_0, \{w_0\}), \dots), \{w_k\})\}$;
- for all $X \subseteq \Sigma$ with $X \notin \{X_1, \dots, X_m\}$, we have $f(X) = \emptyset$.

As an example, the end-component decomposition that corresponds to the UVW in the middle part of Fig. I is a function f with $f(\bar{b}) = \Sigma^* a(\bar{b})^*$, $f(\bar{c}) = \Sigma^* a(\bar{b})^* b(\bar{c})^*$, and $f(X) = \emptyset$ for $X \neq \bar{b}$ and $X \neq \bar{c}$. The decomposition is not maximal as, for example, the word $\{a\}\emptyset^\omega$ is not in the language of the automaton, but we have $\{a\} \notin f(\{\emptyset\}) = \emptyset$.

By the definition of corresponding end-components, every separated UVW has one unique corresponding end-component decomposition. On the other hand, every language has one maximal end-component decomposition. The key result that allows us to reduce finding a UVW for a given language to a problem on finite words combines these two facts:

Lemma 1. *Let L be a language that is representable by a universal very-weak automaton. Then, L is also representable as a separated UVW whose corresponding end-component decomposition is the maximal end-component decomposition of L .*

Proof. Let a UVW be given whose end-component decomposition f is not maximal. The decomposition can be made maximal by taking $f'(X) = \bigcup_{X' \supseteq X} f(X')$ for every $X \subseteq \Sigma$, without changing the language. Building a corresponding UVW only requires taking disjunctions of parts of the original UVW. Since we know that UVW are closed under disjunction, it is assured that there also exists a UVW that corresponds to f' . \square

Thus, in order to obtain a UVW for a given language $L \subset \Sigma^\omega$, we can compute the maximal end-component decomposition f' of L , and for every end component $X \subseteq \Sigma$, compute a non-deterministic very-weak automaton over finite words for $f'(X)$.

Starting with an LTL formula, we can thus translate it to a UVW (if possible) as follows: first of all, we translate the LTL formula to a deterministic Büchi automata (see, e.g., [9] for an overview). Note that as the expressivities of LTL and deterministic Büchi automata are incomparable, this is not always possible. If no translation exists, we however know that there also exists no UVW for the LTL formula, as all languages representable by UVW are also representable by deterministic Büchi automata. After we have obtained the Büchi automaton, we compute for every possible end-component $X \subseteq \Sigma$ from which states S in the automaton every word ending with X^ω is rejected. This is essentially a model checking problem over an automaton with Büchi acceptance condition. This way, for each end component, a deterministic automaton over finite words with S as the set of accepting states then represents the prefix language.

3.2 Decomposing a Language over Finite Words into a Non-deterministic Very-Weak Automaton

This problem of deciding whether there exists a non-deterministic very-weak automaton for a language over finite words is widely studied in the literature. However, constructive algorithms that compute such an automaton are unknown. Hashiguchi studied a more general version of the problem in [19]. His solution is based on computing the maximal distance of an accepted word in a distance automaton. Bojańczyk [4] recently gave a simpler algorithm.

Here, we build on the classical construction by Hashiguchi and modify it in order to be constructive. We describe an iterative algorithm that successively searches for vermicelli in the language to be analyzed. In a nutshell, this is done by searching for accepting words of minimal distance in a distance automaton. Whenever a new vermicelli is found, the automaton is modified in order not to accept words that are already covered by vermicelli that have been found before. At the same time, the new vermicelli can be read from the state sequence in the accepting run. The distance automaton is built as follows.

Definition 4. *Given a DFA $\mathcal{A} = (Q^{\mathcal{A}}, \Sigma, Q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, F^{\mathcal{A}})$ for the language to be analyzed and a NVWF $\mathcal{B} = (Q^{\mathcal{B}}, \Sigma, Q_0^{\mathcal{B}}, \delta^{\mathcal{B}}, F^{\mathcal{B}})$ for the vermicelli already found, the non-deterministic vermicelli-searching distance automaton over finite words $\mathcal{D} = (Q, \Sigma, Q_0, \delta, F)$ is defined as follows:*

$$\begin{aligned}
Q &= 2^{Q^A} \times 2^\Sigma \times \mathbb{B} \times 2^{Q^B} \\
Q_0 &= \{(Q_0^A, \emptyset, \text{false}, Q_0^B)\} \\
F &= \{(S, X, b, R) \mid S \subseteq F^A, (R \cap F^B) = \emptyset\} \\
\delta((S, X, b, R), x) &= \{((S, X, b, R'), 0) \mid R' = \hat{\delta}^B(R, \{x\}), x \in X, b = \text{true}\} \\
&\cup \{((S', X', \text{true}, R'), 1) \mid R' = \hat{\delta}^B(R, \{x\}), x \in X', S' = \hat{\delta}^{A,*}(S, X')\} \\
&\cup \{((S', X', \text{false}, R'), 1) \mid R' = \hat{\delta}^B(R, \{x\}), x \in X', S' = \hat{\delta}^A(S, X')\} \\
&\text{for all } (S, X, b, R) \in Q, x \in \Sigma
\end{aligned}$$

The states in a vermicelli-searching automaton \mathcal{D} are four-tuples (S, X, b, R) such that X and b represent an element in a vermicelli, where b tells us if the current vermicelli element X is starred. During a run, we track in S in which states in \mathcal{A} we can be in after reading some word that is accepted by a vermicelli represented by the vermicelli elements observed in the X and b state components along the run of \mathcal{D} so far. Whenever we have $S \subseteq F^A$, then we know that all these words are accepted by \mathcal{A} . At the same time, the R component simulates all runs of the NVWF \mathcal{B} , and the definition of F ensures that no word that is in the language of \mathcal{B} is accepted by \mathcal{D} . Thus, \mathcal{D} can only find vermicelli that contain some word that is not accepted by \mathcal{B} in their language. Transitions with cost 1 represent moving on to the next vermicelli element.

Theorem 1. *Let \mathcal{A} be an DFA, \mathcal{B} be a NVWF and \mathcal{D} be the corresponding vermicelli-searching distance automaton. We have:*

- $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B})$
- Let $\mathcal{L}(\mathcal{A})$ contain a vermicelli $V = A_1 \dots A_k$, where every A_i is either of the form X^* or X for some $X \subseteq \Sigma$. If V is not covered by $\mathcal{L}(\mathcal{B})$, then \mathcal{D} accepts some word w that is a model of V with a run of distance k . Along this run, the first three state components only change during transitions with a cost of 1, and the second and third component in between changes describe the alphabet symbol sets in the vermicelli and whether the vermicelli elements are starred or not.

As a consequence, since every UVW of size n can be described by a set of vermicelli in which each vermicelli is of length at most n , we can compute a UVW representation of \mathcal{A} by using Algorithm 1. Note that the algorithm does not terminate if \mathcal{A} cannot be represented as a very-weak automaton. Since we can however apply the algorithm by Bojańczyk [4] beforehand to verify the translatability, this imposes no problem.

4 Reduction of the Synthesis Problem to Parity Games

In this section, we explain how to reduce the synthesis problem for specifications of the form $\bigwedge_{a \in \text{Assumptions}} a \rightarrow \bigwedge_{g \in \text{Guarantees}} g$ (or shorter, in assumptions \rightarrow guarantees form), for which each of the assumptions and guarantees is in the common fragment of ACTL and LTL, to solving a parity game. We have discussed in the previous section how one assumption or guarantee can be converted to a UVW. As the conjunction of

Algorithm 1. Translating a DFA \mathcal{A} into a non-deterministic very-weak automaton \mathcal{B}

```

1:  $\mathcal{B} = (\emptyset, \Sigma, \emptyset, \emptyset, \emptyset)$ 
2: repeat
3:    $\mathcal{D}$  = vermicelli searching automaton for  $\mathcal{A}$  and  $\mathcal{B}$ 
4:    $r$  = accepting run of minimal distance in  $\mathcal{D}$ 
5:   if  $r$  was found then
6:     Add  $r$  as vermicelli to  $\mathcal{B}$ 
7:   end if
8: until  $\mathcal{L}(\mathcal{D}) = \emptyset$ 

```

two UVW can be taken by just merging the state sets and the initial states, we also know how to compute *one* UVW for *all* of the assumptions and *one* UVW for *all* of the guarantees. So it remains to be discussed how we combine these two UVW into a game that captures the overall specification.

Bloem et al. [1] describe a way to translate a specification of the assumptions→guarantee form, in which all assumptions and guarantees are in form of deterministic Büchi automata, into a three-color parity game. Essentially, the construction splits the process by converting the assumptions and guarantees to a so-called generalized reactivity(1) game, and then modifying the game structure and adjusting the winning condition to three-color parity. When converting the game, assumption and guarantee pointers represent which assumption and guarantee is observed next for satisfaction. The pointers increment one-by-one, which makes the game solving process a tedious task; for example, if it is the last guarantee (in some assumed order) that the system cannot satisfy, then during the game solving process, this information has to be propagated through all the other pointer values before the process can terminate.

As a remedy, we describe an improved construction here, and let the two players set the pointers. This way, the winning player can set the assumption or guarantee pointer to the problematic assumption or guarantee early in the play, which reduces the time needed for game solving. The game only has colors other than 0 for positions of the environment player, and the states are described as six-tuples. The first two tuple components describe in which states the assumption and guarantee UVW are, followed by the assumption and guarantee pointers that are updated by the system and environment players, respectively. The last two components are Boolean flags that describe whether recently, the assumption (guarantee) state that the respective pointer points to has been left, or the system (environment) player has changed her pointer value, respectively, which is then reflected in the color of the game position. On a formal level, the parity game is built as follows:

Definition 5. Let $\mathcal{A}^a = (Q^A, \Sigma, Q_0^A, \delta^A, F^A)$ and $\mathcal{A}^g = (Q^G, \Sigma, Q_0^G, \delta^G, F^G)$ be two UVW that represent assumptions and guarantees, and Σ_I and Σ_O be sets such that $\Sigma = \Sigma_I \times \Sigma_O$. Without loss of generality, let furthermore $F^A = \{1, \dots, m\}$ and $F^G = \{1, \dots, n\}$. We define the induced synthesis game as a parity game $\mathcal{G} = (V_0, V_1, \Sigma_I, \Sigma_O, E_0, E_1, v_0, c)$ with:

$$V_0 = 2^{Q^A} \times 2^{Q^G} \times \{1, \dots, m\} \times \{1, \dots, n\} \times \mathbb{B} \times \mathbb{B}$$

$$V_1 = V_0 \times \Sigma_I$$

$$E_0 = \{((S^A, S^G, d^A, d^G, b^A, b^G), x) \mapsto (S^A, S^G, d^A, d'^G, \mathbf{false}, b'^G, x) \mid x \in \Sigma_I, \\ (d^G = d'^G) \vee (b'^G = \mathbf{true})\}$$

$$E_1 = \{((S^A, S^G, d^A, d^G, b^A, b^G, x), y) \mapsto (S'^A, S'^G, d'^A, d^G, b'^A, b'^G) \mid y \in \Sigma_O, \\ S'^A = \delta^A(S^A, (x, y)), S'^G = \delta^G(S^G, (x, y)), \\ b'^G = (b^G \vee (d^G \notin S'^G) \vee (d^G \notin \delta^G(d^G, (x, y)))), \\ b'^A = ((d^A \neq d'^A) \vee (d^A \notin S^A) \vee (d^A \notin \delta^A(d^A, (x, y))))\}$$

$$v_0 = (Q_0^A, Q_0^G, 1, 1, \mathbf{false}, \mathbf{false})$$

$$c = \{V_1 \cup \{(q^A, q^G, d^A, d^G, b^A, b^G) \mid \neg b^A \wedge \neg b^G\} \mapsto 0, \{(q^A, q^G, d^A, d^G, \\ b^A, b^G) \mid b^A \wedge \neg b^G\} \mapsto 1, \{(q^A, q^G, d^A, d^G, b^A, b^G) \mid b^G\} \mapsto 2\}$$

For the central correctness claim of this construction, we need some more notation. Given a play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \dots$ for a decision sequence $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \dots$ in the game, we say that a state $q \in Q^A$ is *left* at position $k \in \mathbb{N}$ if for $\pi_k^1 = (S_1^A, S_1^G, d_1^A, d_1^G, b_1^A, b_1^G, \rho_{k-1}^0)$ and $\pi_{k+1}^0 = (S_2^A, S_2^G, d_2^A, d_2^G, b_2^A, b_2^G)$, we have $q \notin S_1^A$ or $q \notin \delta(q, (\rho_k^0, \rho_k^1))$. The construction of \mathcal{G} assures that this is the case whenever any run of the assumption automaton corresponding to the first k choice pairs in the decision sequence leaves state q in the $k+1$ th round or is not in state q in the k th round. The case for the guarantee automaton is analogous. We say that a player *rotates* through the possible pointer values if whenever the state that the pointer refers to is left, the player increases it to the next possible value. In case the highest value is reached, the pointer is set to 1 instead.

Theorem 2. *Let \mathcal{A}^a and \mathcal{A}^g be two UVWs over the alphabet $\Sigma = \Sigma_I \times \Sigma_O$, and \mathcal{G} be the induced synthesis game by Def. 5. The winning strategies for player 1 ensure that along any decision sequence that corresponds to the strategy and in which the input player rotates through the guarantee pointer values, either the sequence is not accepted by \mathcal{A}^a , or the sequence is accepted by \mathcal{A}^g . Furthermore, every Mealy machine with the input Σ_I and output Σ_O for which along any of its runs, the run is either rejected by \mathcal{A}^a or accepted by \mathcal{A}^g induces a winning strategy in \mathcal{G} by having player 1 rotate through the possible assumption pointer values.*

The main message of Theorem 2 is that the games built according to Def. 5 are suitable for solving the synthesis task. Note that there are plays in the game that are winning for the system player, but do not correspond to words that are models of the specification. The reason is that the environment player is not forced to iterate infinitely often over every possible pointer value for the final states of the guarantee automaton. Thus, a winning strategy for the system player in this game does not correspond one-to-one to a Mealy machine that satisfies the specification. For obtaining an implementation for the specification, we need to apply some post-processing to a system player's winning strategy in the parity game.

The post-processing step is however not difficult: observe that the worst case for the system player is that the environment player cycles through the guarantee pointer values. This way, the system player can only win if the decision sequence in the game

represents a model of the guarantees, or the system player is able to eventually point out a rejecting state of the assumption automaton that is never left again. In both cases, the specification is met. Thus, if we attach a round-robin counter for the assumption pointer to a system player's strategy, we obtain a valid result for our synthesis problem.

5 Solving Parity Games Symbolically

For an efficient implementation of the synthesis approach in this paper, the ability to perform the *symbolic* solution of the parity game built according to the construction of the previous section is imperative.

For the scope of this paper, we use a simple parity game solving algorithm that is based on a fixed-point characterization of the winning set of positions in the game, i.e., the positions from which, if the game is started there, the system player can win the game. This approach has three advantages over the classical parity game solving algorithms by Jurdzinski [22] or McNaughton [26]. First of all, it is simpler. Second, it allows applying a nested fixed-point computation acceleration method by Browne et al. [5] that essentially reduces the solution complexity to quadratic time (in the number of game positions), which speeds up the game solving process in contrast to McNaughton's algorithm. Finally, the three-color parity game acceleration method for Jurszinski's algorithm by de Alfaro and Faella [8] is in some sense included for free. Their technique searches for gaps in counter values for visits to positions with color 1. These counters are an artifact that is introduced by Jurdzinski's algorithm. The gaps witness the case that the game solving process can be terminated before the convergence of the counter values. As we do not need such counters here, our algorithm can terminate early automatically without the need to search for such gaps. At the same time, we still have a quadratic complexity of the game solving process. This advantage would also generalize to more than three colors, which the acceleration method in [8] does not.

For the special case of the games in this paper (with only player 0 having colors other than 0 and having only three colors in total), a characterization of the winning positions in a parity game by Emerson and Jutla [14] reduces to the following fixed-point equation:

$$W_0 = \nu X_2. \mu X_1. \nu X_0. (V_1 \cap \diamond X_0) \cup (V_0 \cap C_0 \cap \square X_0) \cup (V_0 \cap C_1 \cap \square X_1) \cup (V_0 \cap C_2 \cap \square X_2)$$

In this formula, C_i represents the set of positions with color i (for every $0 \leq i < 2$), and $\square Y$ and $\diamond Y$ describe, for every $Y \subseteq V$, the set of positions of player 0/player 1 from which player 1 can ensure that after the next move, a position in Y is reached, respectively. All of the operations needed to evaluate this formula can be performed symbolically [6]. Also, encoding the state space of the game into BDDs is not difficult: we can simply assign one bit to every state in the assumption and guarantee automata, one bit for every input or output atomic proposition, two bits for the “recently visited” flags in the game, and $\lceil \log_2 m \rceil + \lceil \log_2 n \rceil$ bits for the pointers.

It remains to be discussed how a winning strategy can be computed symbolically after the sets of winning positions for the two players have been identified. First of all, for $\psi = (V_1 \cap \diamond X_0) \cup (V_0 \cap C_0 \cap \square X_0) \cup (V_0 \cap C_1 \cap \square X_1) \cup (V_0 \cap C_2 \cap \square X_2)$, we compute a sequence of prefixed points $Y_i = \nu X_2. \mu^i X_1. \nu X_0. \psi$ for $i \in \mathbb{N}$. Then,

we take the transition function E_1 of the game and restrict it such that only actions that result in ensuring that the successor position is in the set Y_i for the lowest possible value of i are taken. Any positional strategy that adheres to the restricted transition function is guaranteed to be winning for the system player.

6 Experimental Results

To evaluate the new synthesis approach presented in this paper, it has been implemented in a prototype tool-chain, written in C++ and Python. For the symbolic computation steps, the BDD library CUDD v.2.4.2 [29] was employed. The first step in the tool-chain is to apply the LTL-to-Büchi converter `ltl2ba v.1.1` [18] to the negation of all assumptions and guarantees of the specification. If the result happens to be very-weak, we already have a UVW for the specification part. All remaining assumptions and guarantees are first converted to deterministic Rabin automata using `ltl2dstar v.0.5.1` [23], then translated to equivalent deterministic Büchi automata (if possible), and finally, after a quick check with the construction by Bojańczyk [4] that they represent languages in the common fragment of ACTL and LTL, translated to sets of vermicelli using the construction from Sect. 3. Whenever one of these translations is found to be not possible for some assumption or guarantee, the specification is known not to lie in the supported specification fragment and rejected. The construction from Sect. 3 is performed symbolically, using BDDs and dynamic variable reordering for the BDD variables. The UVW for the individual assumptions and guarantees are then merged and some simulation-based automaton minimization steps are applied. In contrast to general bisimulation-based minimization techniques for non-deterministic Büchi automata (see, e.g., [15]), we make use of the fact that the automata are very-weak, which allows applying more optimizations. The optimization steps are:

- States that are reached by the same set of prefix words are merged (unless this would introduce a loop).
- States with the same language are merged.
- For every pair of states (q, q') in the automaton, if q is reached by at least as many prefix words as q' , but q' has a greater language than q , we remove q' .

Finally, we perform symbolic parity game solving for the synthesis game build using the minimized UVW for the assumptions and guarantees as described in Sect. 4 and Sect. 5. In case of realizability, we use an algorithm by Kukula and Shiple [24] to compute a circuit description of the implementation. The prototype tool also checks for which input/output bits it makes sense to encode the last values into the game as an additional component. This can happen if there are many states in the UVW for which it only depends on the last input and output whether we are in that state at a certain time. Then, we can save the BDD bits for these states. For checking the resulting implementations for correctness, we use NuSMV v.2.5.4 [7].

All computation times given in the following were obtained on an Intel Core 2 Duo (1.86 Ghz) computer running Linux. All tools considered are single-threaded. We restricted the memory usage to 2 GB and set a timeout of 3600 seconds. We compare our new approach against Acacia+ v.1.2 [16][17] and Unbeast v.0.6 [13], both

using `ltl2ba`. Both synthesis tools implement semi-algorithms, i.e., we need to test for realizability and unrealizability separately and only give the computation time of the invocation that terminated for comparison. We could not compare against tools that implement generalized reactivity(1) synthesis such as Anzu [21] as due to the non-standard semantics (see [11], p. 4 for details) used there, the results would not be meaningful.

Benchmarks

First of all, we consider the load balancer from [10]. This benchmark is for synthesis tools that are capable of handling full LTL, and consists of 10 scalable specifications. Out of these, we found 6 to be contained in the supported fragment by our approach, including the final specification of the load balancer. Table I summarizes the results. It can be observed that the two synthesis tools for full LTL are clearly outperformed on the supported specifications.

As a second benchmark, we use the non-pre-synthesized AMBA high performance bus arbiter specification described in [2], which is again scalable in the number of clients. Here, our tool-chain is able to synthesize the two-client version in 151 seconds, while the three-client version takes 1422 seconds. In both cases, most of the time is spent on the symbolic game solving step. Neither `Unbeast` nor `Acacia+` can handle any of these two cases within 1 hour of computation time. According to [3], with the pre-synthesized version of the specification of [2], the generalized reactivity(1) tool used in the experimental evaluation of [3] could only handle up to four clients. Thus, our approach comes close in terms of efficiency, but without the need of pre-synthesis. For completeness, it must be added, however, that a (manual) rewriting of the specification was later able boost the generalized reactivity(1) synthesis performance [3] on this benchmark.

Table 1. Running times of the synthesis tools `Acacia` (“A”), `Unbeast` (“U”) and a prototype tool for the approach presented in this paper (“B”) for the load balancer benchmark, using setting labels from [10]. For each combination of assumptions and guarantees, it is reported whether the specification was realizable (+/-) and how long the computation took (in seconds).

Tool	Setting / # Clients	2	3	4	5	6	7	8	9
B	1	+ 0.3	+ 0.4	+ 0.4	+ 0.4	+ 0.5	+ 0.5	+ 0.6	+ 0.6
		+ 0.0	+ 0.0	+ 0.6	+ 0.0	+ 0.0	+ 0.0	+ 0.1	+ 0.2
		+ 0.3	+ 0.3	+ 0.3	+ 0.3	+ 0.4	+ 0.4	+ 0.4	+ 0.5
B	1 \wedge 2	+ 0.4	+ 0.4	+ 0.4	+ 0.5	+ 0.6	+ 0.9	+ 2.2	+ 6.9
		+ 0.7	+ 0.0	+ 0.1	+ 0.1	+ 0.1	+ 0.1	+ 0.2	+ 0.3
		+ 0.3	+ 0.4	+ 1.2	+ 0.3	+ 0.4	+ 0.7	+ 1.8	+ 5.5
B	1 \wedge 2 \wedge 3	- 0.5	- 0.6	- 0.7	- 0.9	- 1.2	- 1.7	- 3.4	- 7.6
		- 0.0	- 0.0	- 0.1	- 0.1	- 0.2	- 1.3	- 11.5	- 145.4
		- 0.3	- 0.3	- 0.4	- 2.9	timeout	timeout	timeout	timeout
B	6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8	+ 0.6	+ 0.8	+ 0.9	+ 1.2	+ 1.6	+ 2.2	+ 4.0	+ 9.7
		+ 0.1	+ 0.4	+ 1.4	+ 39.9	timeout	timeout	timeout	timeout
		+ 2.1	+ 1.3	timeout	timeout	timeout	timeout	timeout	timeout
B	6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9	- 0.7	- 0.9	- 1.2	- 1.6	- 2.1	- 3.2	- 5.5	- 11.5
		- 0.0	- 0.1	- 0.2	- 1.4	- 28.5	- 886.4	timeout	timeout
		- 0.4	- 0.4	- 2.6	timeout	timeout	timeout	timeout	timeout
B	6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9	+ 0.8	+ 1.0	+ 1.3	+ 2.3	+ 2.5	+ 3.3	+ 5.7	+ 11.8
		+ 0.3	+ 2.2	+ 23.7	+ 632.5	timeout	timeout	timeout	timeout
		+ 0.9	+ 0.8	+ 16.3	timeout	timeout	timeout	timeout	timeout

7 Conclusion

In this paper, we have proposed ACTL \cap LTL as a specification fragment that combines expressivity and efficiency for the synthesis of reactive systems. We gave novel algorithms and constructions for the individual steps in the synthesis workflow. In particular, we gave the first procedure to obtain universal very-weak automata from LTL formulas (if possible) and described a novel procedure for building a parity game from assumption and guarantee properties that speeds up the game solving process by letting the two players choose the next obligations to the respective other player in the game.

We did not fully exploit the favorable properties of UVW in the paper, and only see the experimental evaluation herein as a start. For example, since in the structure of the game built from UVWs, we keep track of in which assumption and guarantee states we could be in, the game lends itself to the symbolic encoding of the prefixed points in the game solving process using anti-chains [16].

Also, the approach can easily be extended to support properties whose *negation* is in the common fragment of ACTL and LTL. This would allow using *persistence* properties like “the system must eventually signal readiness forever”. We recently described in [12] how generalized reactivity(1) synthesis can be extended to handle such properties, resulting in five-color parity games. The constructions in this paper are easy to extend accordingly.

Acknowledgements. The author wants to thank Bernd Finkbeiner and Sven Schewe for the interesting discussions in which the idea of Sect. 4 was developed. Furthermore, the author wants to thank Armin Biere for pointing out the common fragment of ACTL and LTL as potentially interesting for synthesis.

References

1. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B.: Robustness in the Presence of Liveness. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 410–424. Springer, Heidelberg (2010)
2. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Interactive presentation: Automatic hardware synthesis from specifications: a case study. In: Lauwersins, R., Madsen, J. (eds.) DATE, pp. 1188–1193. ACM (2007)
3. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. Electr. Notes Theor. Comput. Sci. 190(4), 3–16 (2007)
4. Bojańczyk, M.: The Common Fragment of ACTL and LTL. In: Amadio, R.M. (ed.) FOS-SACS 2008. LNCS, vol. 4962, pp. 172–185. Springer, Heidelberg (2008)
5. Browne, A., Clarke, E.M., Jha, S., Long, D.E., Marrero, W.R.: An improved algorithm for the evaluation of fixpoint expressions. Theor. Comput. Sci. 178(1-2), 237–255 (1997)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. 98(2), 142–170 (1992)
7. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)

8. de Alfaro, L., Faella, M.: An Accelerated Algorithm for 3-Color Parity Games with an Application to Timed Games. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 108–120. Springer, Heidelberg (2007)
9. Ehlers, R.: Minimising Deterministic Büchi Automata Precisely Using SAT Solving. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 326–332. Springer, Heidelberg (2010)
10. Ehlers, R.: Symbolic Bounded Synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 365–379. Springer, Heidelberg (2010)
11. Ehlers, R.: Experimental aspects of synthesis. In: Reich, J., Finkbeiner, B. (eds.) iWIGP. EPTCS, vol. 50, pp. 1–16 (2011)
12. Ehlers, R.: Generalized Rabin(1) Synthesis with Applications to Robust System Synthesis. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 101–115. Springer, Heidelberg (2011)
13. Ehlers, R.: Unbeast: Symbolic Bounded Synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
14. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: FOCS, pp. 368–377. IEEE Computer Society (1991)
15. Etessami, K., Wilke, T., Schuller, R.A.: Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 694–707. Springer, Heidelberg (2001)
16. Filiot, E., Jin, N., Raskin, J.-F.: An Antichain Algorithm for LTL Realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
17. Filiot, E., Jin, N., Raskin, J.-F.: Compositional Algorithms for LTL Synthesis. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 112–127. Springer, Heidelberg (2010)
18. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
19. Hashiguchi, K.: Representation theorems on regular languages. *J. Comput. Syst. Sci.* 27(1), 101–115 (1983)
20. Janin, D., Lenzi, G.: On the relationship between monadic and weak monadic second order logic on arbitrary trees, with applications to the mu-calculus. *Fundam. Inform.* 61(3–4), 247–265 (2004)
21. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A Tool for Property Synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
22. Jurdziński, M.: Small Progress Measures for Solving Parity Games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
23. Klein, J., Baier, C.: Experiments with deterministic ω -automata for formulas of linear temporal logic. *Theor. Comput. Sci.* 363(2), 182–195 (2006)
24. Kukula, J.H., Shiple, T.R.: Building Circuits from Relations. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 113–123. Springer, Heidelberg (2000)
25. Maidl, M.: The common fragment of CTL and LTL. In: FOCS, pp. 643–652 (2000)
26. McNaughton, R.: Infinite games played on finite graphs. *Ann. Pure Appl. Logic* 65(2), 149–184 (1993)
27. 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 (2006)
28. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD, pp. 77–84. IEEE (2009)
29. Somenzi, F.: CUDD: CU decision diagram package, release 2.4.2 (2009)

Learning Boolean Functions Incrementally*

Yu-Fang Chen and Bow-Yaw Wang

Academia Sinica, Taiwan

Abstract. Classical learning algorithms for Boolean functions assume that unknown targets are Boolean functions over fixed variables. The assumption precludes scenarios where indefinitely many variables are needed. It also induces unnecessary queries when many variables are redundant. Based on a classical learning algorithm for Boolean functions, we develop two learning algorithms to infer Boolean functions over enlarging sets of ordered variables. We evaluate their performance in the learning-based loop invariant generation framework.

1 Introduction

Algorithmic learning is a technique for inferring a representation of an unknown target in a specified instance space. When designing a learning algorithm, one formalizes intended scenarios as a learning model. In Boolean function learning, for instance, we are interested in finding a representation (such as a Boolean formula [3]) of an unknown target amongst Boolean functions over fixed variables. The goal of a learning algorithm is to generate a representation of the unknown target under the learning model [113].

Inferring unknown targets over fixed variables however is not realistic in applications such as loop invariant generation [11412], or contextual assumption synthesis [54]. In loop invariant generation, one considers a loop annotated with pre- and post-conditions. The instance space hence consists of quantifier-free formulae over a given set of atomic predicates. We are interested in finding a quantifier-free formula which establishes the pre- and post-conditions in the specified instance space [11412]. Note that the given set of atomic predicates may not be able to express any loop invariant. If the current atomic predicates are not sufficiently expressive, more atomic predicates will be added. Hence the set of atomic predicates is not fixed but indefinite. Yet classical learning presumes a fixed set of variables for unknown targets. It does not consider scenarios where new variables can be introduced on the fly. The classical learning model therefore do not really fit the scenario of loop invariant generation.

Another drawback in classical learning algorithms for Boolean functions is their inefficiency in the presence of redundant variables. In contextual assumption generation, one considers the problem of verifying a system composed of two components. We would like to replace one of the components by a contextual assumption so as to verify the system more efficiently. The instance space therefore consists of transition relations over model variables. We are interested in finding the transition relation of a contextual assumption that solves the verification problem [54]. Observe that a contextual

* This work is partially supported by the National Science Council of Taiwan under the grant numbers 99-2218-E-001-002-MY3 and 100-2221-E-002-116-.

assumption is synthesized for a specific verification problem. If a model variable is not relevant to the problem, contextual assumptions can safely ignore it. Thus we are looking for an unknown transition relation over a subset of model variables. One would naturally expect a learning algorithm to perform really well when many model variables are irrelevant. Yet the complexity of classical learning algorithms depends on the number of given variables, not relevant ones. Classical learning can be unexpectedly inefficient when many given variables are redundant.

We propose to infer Boolean functions over indefinitely many variables by incremental learning. Instead of Boolean functions over a fixed number of variables, we infer the unknown target by enlarging sets of *ordered* variables incrementally. At iteration ℓ , we try to infer the unknown target as a Boolean function over the first ℓ variables. Our incremental learning algorithm terminates if it infers the target. Otherwise, it proceeds to the next iteration and tries to infer the unknown target as a Boolean function over the first $\ell + 1$ variables. Since the unknown target is over finitely many variables, our incremental learning algorithm will infer the target after finitely many iterations.

A naive approach to incremental learning is to apply the classical CDNF learning algorithm for Boolean functions at each iteration. The simple approach however does not work. Note that the complexity of the CDNF algorithm depends on the formula size of the unknown target. When targets are arbitrary, their formula sizes are exponential in the number of variables. Since $\Omega(2^\ell)$ queries are needed to infer an arbitrary target over ℓ variables in the worst case, the naive algorithm has to make as many queries before it gives up the iteration ℓ . Subsequently, the naive algorithm would require an exponential number of queries for *every* unknown target and could not be efficient.

To solve this problem, we develop a criterion to detect failures at each iteration dynamically. At iteration ℓ , our incremental algorithm checks whether the unknown target is a Boolean function over the first ℓ variables during the course of inference. If the incremental algorithm detects that the target needs more than the first ℓ variables, the iteration ℓ is going to fail. Hence the incremental learning algorithm should abort and proceed to the next iteration. We propose two incremental learning algorithms with dynamic failure detection. In our simple incremental learning algorithm CDNF+, the classical learning algorithm is initialized at each iteration. Information from previous iterations hence is lost. Our more sophisticated incremental learning algorithm CDNF++ retains such information and attains a better complexity bound. Under a generalized learning model, both of our incremental algorithms require at most a polynomial number of queries in the formula size and the number of ordered variables in the target. Incremental learning on certain Boolean functions is still feasible.

To attest the performance of our incremental learning algorithms for Boolean functions, we compare with the classical algorithm in the learning-based loop invariant generation framework [11][14][12]. To evaluate the performance of incremental learning in typical settings, we consider a simple heuristic variable ordering from the application domain. Our incremental learning algorithms achieve up to 59.8% of speedup with the heuristic ordering. To estimate the worst-case performance of incremental learning, we adopt random variable orderings instead of the heuristic ordering. Excluding one extreme case, the incremental learning algorithms perform slightly better than the classical algorithm with random orderings. Since a sensible variable ordering can often

be chosen by domain experts in most applications, the artificial worst-case scenario is unlikely to happen. We therefore expect our new algorithms to prevail in practice.

In the classical CDNF learning algorithm for Boolean functions, unknown targets are Boolean functions over fixed variables [3]. It is not applicable to scenarios where unknown targets are over indefinitely many variables. Combining with predicate abstraction and decision procedures, the CDNF algorithm is used to generate invariants for annotated loops [11][14][12], and transition invariants for termination analysis [16]. The classical algorithm is also deployed in assume-guarantee reasoning to infer contextual assumptions automatically [54]. In these applications, the CDNF algorithm is used as a black box. We propose a new learning model and develop incremental algorithms under the new model. We do not know of any learning algorithm for Boolean functions over indefinitely many variables. Abstraction techniques in regular language learning are seemingly relevant [8][2][10]. Recall that the L^* algorithm does not apply when queries are answered nondeterministically. It is necessary to bring the learning algorithm to consistent states upon nondeterministic answers induced by abstraction. Incremental queries can introduce inconsistencies. We also have to bring the incremental learning algorithms back to consistent states. Since this work is about learning Boolean functions, it is related to [8][2][10] only in spirits. Many applications of the L^* algorithm for regular languages have been proposed (see [9], for example).

This paper is organized as follows. After Introduction, preliminaries and notations are given in Section 2. We then review the CDNF algorithm (Section 3). Section 4 presents our technical contribution. It is followed by experimental results in Section 5. Finally, Section 6 concludes our presentation.

2 Preliminary

Let $\mathbb{B} = \{\perp, \top\}$ be the *Boolean domain* and $\mathbf{x} = \{x_1, x_2, \dots, x_n, \dots\}$ an infinite set of ordered Boolean variables. We write \mathbf{x}_ℓ for the subset $\{x_1, x_2, \dots, x_\ell\}$ of \mathbf{x} . A *valuation* over \mathbf{x}_ℓ is a function from \mathbf{x}_ℓ to \mathbb{B} . The set of all valuations over \mathbf{x}_ℓ is denoted by Val_ℓ . For any valuation $u \in Val_\ell$, $x \in \mathbf{x}_{\ell+1}$, and $b \in \mathbb{B}$, define

$$u[x \mapsto b](y) = \begin{cases} u(y) & \text{if } y \neq x \\ b & \text{if } y = x. \end{cases}$$

Note that $u[x_{\ell+1} \mapsto b] \in Val_{\ell+1}$ for every $u \in Val_\ell$. Let $\perp_\ell \in Val_\ell$ be the valuation mapping every $x \in \mathbf{x}_\ell$ to \perp , and the valuation $\top_\ell \in Val_\ell$ mapping every $x \in \mathbf{x}_\ell$ to \top . The *projection* of a valuation v on \mathbf{x}_ℓ is the valuation $u \in Val_\ell$ such that $u(x) = v(x)$ for every $x \in \mathbf{x}_\ell$. The symbol \oplus stands for the component-wise exclusive-or operator. Thus $u \oplus \perp_\ell = u$ for every $u \in Val_\ell$. If $R \subseteq Val_\ell$ is a set of valuations and $u \in Val_\ell$, we define $R \oplus u = \{r \oplus u : r \in R\}$. Thus $R \oplus \perp_\ell = R$ for every $R \subseteq Val_\ell$. A *Boolean function* over \mathbf{x}_ℓ is a mapping from Val_ℓ to \mathbb{B} . Let f be a Boolean function. For any valuation $u \in Val_\ell$, the notation $f(u)$ denotes the Boolean function obtained by assigning x to $u(x)$ in f . Particularly, $f(u)$ is the Boolean outcome of f on any valuation $u \in Val_\ell$ when f is a Boolean function over \mathbf{x}_ℓ . Moreover, we say u is a *satisfying valuation* of the Boolean function f if $f(u) = \top$; it is an *unsatisfying valuation* of f if $f(u) = \perp$. When there is a satisfying valuation of a Boolean function

f , we say f is *satisfiable*. A Boolean formula F over \mathbf{x}_ℓ represents a Boolean function $\llbracket F \rrbracket_\ell$ defined as follows. On any valuation $u \in \text{Val}_\ell$, $\llbracket F \rrbracket_\ell(u)$ is obtained by evaluating F under the valuation u . For example, $\llbracket x_1 \implies x_2 \rrbracket_2(\perp_2) = \top$.

A *literal* is a Boolean variable or its negation. A *term* is a conjunction of literals. A *clause* is a disjunction of literals. A Boolean formula is in *disjunctive normal form* (DNF) if it is a disjunction of terms. A Boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. A formula in CNF (DNF) is a *CNF* (DNF, respectively) *formula*. A Boolean formula is in *conjunctive disjunctive normal form* (CDNF) if it is a conjunction of DNF formulae. A formula in CDNF is a *CDNF formula*.

3 The CDNF Algorithm

The CDNF algorithm is an exact learning algorithm for Boolean functions over \mathbf{x}_n [3]. Suppose f is an unknown *target* Boolean function over \mathbf{x}_n . The learning algorithm infers a CDNF formula representing f by interacting with a teacher. The *teacher* is responsible for answering two types of queries.

- *Membership queries* $\text{MEM}_n(v)$ with $v \in \text{Val}_n$. If $f(v) = \top$, the teacher answers YES; otherwise, she answers NO.
- *Equivalence queries* $\text{EQ}_n(F)$ with a Boolean formula F over \mathbf{x}_n as the *conjecture*. If $\llbracket F \rrbracket_n = f$, the teacher answers YES. Otherwise, the teacher returns a *counterexample* $v \in \text{Val}_n$ such that $\llbracket F \rrbracket_n(v) \neq f(v)$.

Let $v \in \text{Val}_n$ be a valuation and F a Boolean formula over \mathbf{x}_n . We write $\text{MEM}_n(v) \rightarrow Y$ and $\text{EQ}_n(F) \rightarrow Z$ to denote that Y and Z are the answers to the membership query on v and equivalence query on F , respectively.

```

1  $t \leftarrow 0;$ 
2  $\text{EQ}_n(\text{true}) \rightarrow v;$ 
3 if  $v$  is YES then return true;
4  $t \leftarrow t + 1;$ 
5  $H_t, R_t, a_t \leftarrow \text{false}, \emptyset, v;$  // assert  $\text{MEM}_n(a_t) \rightarrow NO$ 
6  $\text{EQ}_n(\wedge_{i=1}^t H_i) \rightarrow v;$ 
7 if  $v$  is YES then return  $\wedge_{i=1}^t H_i$ ;
8  $I \leftarrow \{i : \llbracket H_i \rrbracket_n(v) = \perp\};$ 
9 if  $I = \emptyset$  then goto 4
10 foreach  $i \in I$  do
11    $r \leftarrow \text{walkTo}(n, a_i, v);$  // assert  $\text{MEM}_n(r) \rightarrow YES$ 
12    $R_i \leftarrow R_i \cup \{r\};$ 
13 end
14 foreach  $i = 1, \dots, t$  do  $H_i \leftarrow M_{\text{DNF}}(R_i \oplus a_i)(\mathbf{x}_n \oplus a_i);$ 
15 goto 4

```

Algorithm 1. The CDNF Algorithm

We reprint the CDNF algorithm in Algorithm 1. In the algorithm, conjectures in equivalence queries are always CDNF formulae. The variable t maintains the number of DNF formulae in the current conjecture. Initially, the variable t is set to 0. The conjecture is hence degenerated to true (line 2, Algorithm 1).

Three variables keep track of each DNF formula in the conjecture. For the i -th DNF formula, the variable a_i is a valuation over \mathbf{x}_n , the variable R_i is a set of valuations over \mathbf{x}_n , and the variable H_i is a DNF formula over \mathbf{x}_n . The i -th DNF formula H_i is derived from a_i and R_i by M_{DNF} (line 14, Algorithm 1):

$$M_{\text{DNF}}(s) = \begin{cases} \bigwedge_{s(x_i)=\top} x_i & \text{if } s \neq \perp_n \\ \text{true} & \text{otherwise} \end{cases} \quad M_{\text{DNF}}(S) = \begin{cases} \bigvee_{s \in S} M_{\text{DNF}}(s) & \text{if } S \neq \emptyset \\ \text{false} & \text{otherwise} \end{cases}$$

For instance, $M_{\text{DNF}}(\{\perp_2, \top_2\}) = M_{\text{DNF}}(\perp_2) \vee M_{\text{DNF}}(\top_2) = \text{true} \vee (x_1 \wedge x_2)$.

When a new DNF formula is added to the conjecture, the variable R_t is the empty set and the variable H_t is set to false accordingly (line 5, Algorithm 1). Conjectures in equivalence queries are conjunctions of H_i 's.

In order to understand our extensions to the CDNF learning algorithm, we give a new characterization of variables associated with the i -th DNF formulae in Algorithm 1. Note that a_i was defined when the i -th DNF formula was created and added to the conjecture (line 5, Algorithm 1). It is not hard to see that a_i is a valuation with $\text{MEM}_n(a_i) \rightarrow \text{NO}$. First, a_1 was a counterexample to the equivalence query $\text{EQ}_n(\text{true})$. We have $\text{MEM}_n(a_1) \rightarrow \text{NO}$. For $i > 1$, observe that a_i was the counterexample to the equivalence query $\text{EQ}_n(\bigwedge_{j=1}^{i-1} H_j)$ (line 6, Algorithm 1). Furthermore, a_i was added when the set $\{j < i : [\![H_j]\!]_n(a_i) = \perp\}$ was empty (line 9, Algorithm 1). Since $[\![\bigwedge_{j=1}^{i-1} H_j]\!]_n(a_i) = \top$ and $\text{EQ}_n(\bigwedge_{j=1}^{i-1} H_j) \rightarrow a_i$, we have $\text{MEM}_n(a_i) \rightarrow \text{NO}$.

The valuations in R_i can be characterized as easily. When the counterexample v to the equivalence query $\text{EQ}_n(\bigwedge_{i=1}^t H_i)$ is returned (line 6, Algorithm 1), the CDNF algorithm checks if the set $\{i : [\![H_i]\!]_n(v) = \perp\}$ is empty (line 9, Algorithm 1). If not, we have $[\![\bigwedge_{i=1}^t H_i]\!]_n(v) = \perp$. Thus $\text{MEM}_n(v) \rightarrow \text{YES}$ for v is a counterexample to $\text{EQ}_n(\bigwedge_{i=1}^t H_i)$. For each i such that $[\![H_i]\!]_n(v) = \perp$, the result of $\text{walkTo}(n, a_i, v)$ is added to R_i (line 12, Algorithm 1). Algorithm 2 gives the details of $\text{walkTo}(\ell, a, v)$.

The algorithm $\text{walkTo}(\ell, a, v)$ finds an $x \in \mathbf{x}_\ell$ with $v(x) \neq a(x)$ and flips the value of $v(x)$. If the new valuation yields YES on a membership query, it continues flipping other values of v different from a . Otherwise, the algorithm reverts to the old value of $v(x)$ and flips another value. Roughly, $\text{walkTo}(\ell, a, v)$ computes a valuation $r \in \text{Val}_n$ closest to a such that $\text{MEM}_n(r) \rightarrow \text{YES}$. Define

$$N_\ell(a, r) = \{w \in \text{Val}_\ell : w = r[x \mapsto a(x)] \text{ where } x \in \mathbf{x}_\ell \text{ and } r(x) \neq a(x)\}.$$

Each valuation in $N_\ell(a, r)$ is obtained by flipping the value of exactly one $x \in \mathbf{x}_\ell$ on r with $r(x) \neq a(x)$. Each valuation in $N_\ell(a, r)$ is thus closer to a than r . The following lemma summarizes Algorithm 2.

Lemma 1. *Let $a, v \in \text{Val}_\ell$ ($1 \leq \ell$) be that $\text{MEM}_\ell(a) \rightarrow \text{NO}$ and $\text{MEM}_\ell(v) \rightarrow \text{YES}$. Assume $r = \text{walkTo}(\ell, a, v)$ (Algorithm 2). Then $\text{MEM}_\ell(r) \rightarrow \text{YES}$, and $\text{MEM}_\ell(w) \rightarrow \text{NO}$ for every $w \in N_\ell(a, r)$.*

```

Input:  $\ell \in \mathbb{N} : 1 \leq \ell; a \in \text{Val}_\ell : \text{MEM}_\ell(a) \rightarrow \text{NO}; v \in \text{Val}_\ell : \text{MEM}_\ell(v) \rightarrow \text{YES}$ 
Output:  $r \in \text{Val}_\ell : \text{MEM}_\ell(r) \rightarrow \text{YES}$ 

1  $r \leftarrow v;$ 
2  $k \leftarrow 1;$ 
3 while  $k \leq \ell$  do
4   if  $r(x_k) = a(x_k)$  then  $k \leftarrow k + 1;$ 
5   else
6      $r(x_k) \leftarrow a(x_k);$ 
7     if  $\text{MEM}_\ell(r) \rightarrow \text{NO}$  then
8        $r(x_k) \leftarrow \neg a(x_k);$ 
9        $k \leftarrow k + 1;$ 
10    else  $k \leftarrow 0;$ 
11 end
12 return  $r;$ 

```

Algorithm 2. $\text{walkTo}(\ell, a, v)$

Recall that R_i consists of the result of $\text{walkTo}(n, a_i, v)$ where $\text{MEM}_n(a_i) \rightarrow \text{NO}$ and $\text{MEM}_n(v) \rightarrow \text{YES}$. Thus $\text{MEM}_n(r) \rightarrow \text{YES}$ for every $r \in R$; $\text{MEM}_n(w) \rightarrow \text{NO}$ for every $r \in R$ and $w \in N_n(a_i, r)$ (Lemma 1). We characterize the pairs (a, R) 's maintained in the learning algorithm with the following definition:

Definition 1. For $a \in \text{Val}_n$ and $R \subseteq \text{Val}_n$, define the property $\Gamma(a, R)$ by

1. $\text{MEM}_n(a) \rightarrow \text{NO};$
2. $\text{MEM}_n(r) \rightarrow \text{YES}$ for every $r \in R$; and
3. $\text{MEM}_n(w) \rightarrow \text{NO}$ for every $r \in R$ and $w \in N_n(a, r)$.

Suppose $\llbracket \neg x_1 \vee \neg x_2 \rrbracket_2$ is the target Boolean function over \mathbf{x}_2 as an example. Let $r(x_1) = \perp$ and $r(x_2) = \top$. We have $\Gamma(\top_2, \{r\})$ but not $\Gamma(\top_2, \{\perp_2\})$.

The following lemma states that $\Gamma(a_i, R_i)$ holds for $1 \leq i \leq t$ in the CDNF algorithm. We call (a, R) a *speculative support* when $\Gamma(a, R)$ holds.

Lemma 2. At line 6 of Algorithm 1, $\Gamma(a_i, R_i)$ holds for every $1 \leq i \leq t$.

The *size* of a DNF formula is the number of terms in the formula; the *size* of a CNF formula is the number of clauses in it. Let f be a Boolean function over \mathbf{x}_n . The *DNF size* of f (denoted by $|f|_{\text{DNF}}$) is the minimal size over all DNF formulae representing f ; the *CNF size* of f (denoted by $|f|_{\text{CNF}}$) is the minimal size of all CNF formulae representing f . The number of speculative supports and the size of R in each speculative support (a, R) give the following bounds.

Theorem 1 ([3]). Let f be an unknown target Boolean function over \mathbf{x}_n . The CDNF algorithm (Algorithm 1) infers f within $O(n^2 |f|_{\text{CNF}} |f|_{\text{DNF}})$ membership and $O(|f|_{\text{CNF}} |f|_{\text{DNF}})$ equivalence queries.

Note that the complexity of the CDNF algorithm is a polynomial in the size of the variable set \mathbf{x}_n . If all but one variables in \mathbf{x}_n are redundant, the learning algorithm still requires $O(n^2)$ membership queries to infer the target.

4 Incremental Learning

The CDNF algorithm infers an unknown target among Boolean functions over a fixed number of variables. It is not applicable to scenarios where targets are Boolean functions over indefinitely many variables. Moreover, the complexity of the CDNF algorithm is a polynomial in the number of given variables. It can be unexpectedly inefficient when many variables are redundant in the unknown target.

It appears that these issues could be resolved by invoking the CDNF algorithm iteratively. A naive incremental learning algorithm adopts the classical learning algorithm to infer the unknown target as a Boolean function over \mathbf{x}_ℓ at iteration ℓ . If it succeeds, the naive algorithm reports the inferred result. Otherwise, the naive algorithm increments the number of variables and invokes the CDNF algorithm to infer the unknown target as a Boolean function over $\mathbf{x}_{\ell+1}$. The naive approach however has two problems.

The first problem is to answer queries. Recall that the teacher knows a target Boolean function over, say, \mathbf{x}_m . At iteration ℓ , the naive incremental algorithm infers the unknown target as a Boolean function over \mathbf{x}_ℓ . It thus makes queries on valuations and conjectures over \mathbf{x}_ℓ . Yet the target Boolean function is over \mathbf{x}_m . It is unclear how the teacher answers queries at iteration ℓ when $\ell \neq m$. A new learning model where the teacher answers such queries is necessary for learning Boolean functions incrementally.

The other problem of the naive approach is its inefficiency. Recall that the complexity of the CDNF algorithm depends on the CNF and DNF sizes of the unknown target. Since targets are arbitrary, $\Omega(2^\ell)$ queries are needed to decide whether the learning algorithm fails to infer the target at iteration ℓ . Deciding failures of inference requires an exponential number of queries at each iteration. Naively adopting the CDNF algorithm would be very inefficient compared to the classical learning algorithm. A more sophisticated mechanism to identify failures of inference at each iteration is indispensable.

For the first problem, we generalize the classical learning model to enable the teacher answering queries at all iterations (Section 4.1). To address the second problem, we develop a criterion for determining failures of inference dynamically and use it in our simple incremental learning algorithm (Section 4.2). A sophisticated incremental algorithm with an economical management of information is presented in Section 4.3.

4.1 Incremental Teacher

Assume a target Boolean function f over a finite subset of \mathbf{x} . In our incremental learning model, an *incremental teacher* should answer the following queries:

- *Incremental membership queries* $MEM_\ell(u)$ with $u \in Val_\ell$. If $f(u)$ is satisfiable, the incremental teacher answers YES; otherwise, she answers NO.
- *Incremental non-membership queries* $\overline{MEM}_\ell(u)$ with $u \in Val_\ell$. If $\neg f(u)$ is satisfiable, the incremental teacher answers YES; otherwise, NO.
- *Incremental equivalence queries* $EQ_\ell(G)$ with a Boolean formula G over \mathbf{x}_ℓ . If $\llbracket G \rrbracket_\ell = f$, the incremental teacher answers YES. Otherwise, she returns the projection of a valuation $v \in Val_\mathbf{x}$ on \mathbf{x}_ℓ where $\llbracket G \rrbracket_\ell(v) \neq f(v) \in \mathbb{B}$.

Example. Let $f = x_1 \oplus x_2$. On incremental queries $MEM_1(\perp\!\!\!\perp_1)$ and $\overline{MEM}_1(\perp\!\!\!\perp_1)$, the incremental teacher answers YES. Similarly, the incremental teacher answers YES

on incremental queries $\text{MEM}_1(\top_1)$ and $\overline{\text{MEM}}_1(\top_1)$. On incremental equivalence queries $\text{EQ}_1(\text{true})$ or $\text{EQ}_1(\text{false})$, \perp is a counterexample.

Incremental queries allow a learning algorithm to acquire (incomplete) information about the unknown target function. Intuitively, the answer to an incremental membership query on a valuation reveals whether a completion of the valuation gives a satisfying valuation; the answer to an incremental non-membership query shows whether a completion gives an unsatisfying valuation. Incremental equivalence queries check whether the target is equivalent to a Boolean formula over specified variables. If not, a valuation differentiates the conjecture and the target. The projection of such a valuation on specified variables is returned as a counterexample. The following lemma is useful.

Lemma 3. *Assume a target Boolean function over \mathbf{x}_m and $1 \leq \ell \leq m$.*

1. *For any valuation $v \in \text{Val}_m$, $\text{MEM}_m(v) \rightarrow \text{YES}$ iff $\overline{\text{MEM}}_m(v) \rightarrow \text{NO}$.*
2. *For any Boolean formula G and valuation u over \mathbf{x}_ℓ , $\llbracket G \rrbracket_\ell(u) = \perp$ and $\text{EQ}_\ell(G) \rightarrow u$ imply $\text{MEM}_\ell(u) \rightarrow \text{YES}$.*
3. *For any Boolean formula G and valuation u over \mathbf{x}_ℓ , $\llbracket G \rrbracket_\ell(u) = \top$ and $\text{EQ}_\ell(G) \rightarrow u$ imply $\overline{\text{MEM}}_\ell(u) \rightarrow \text{YES}$.*

4.2 The CDNF+ Algorithm

Suppose that the CDNF algorithm is inferring an unknown target as a Boolean function over \mathbf{x}_ℓ at iteration ℓ . We check if the classical algorithm will fail at this iteration. If so, we abort and re-instantiate the CDNF algorithm to infer the unknown target as a Boolean function over $\mathbf{x}_{\ell+1}$ at the next iteration. To determine failures of inference, recall that the CDNF algorithm is exact. If the unknown target is indeed a Boolean function over \mathbf{x}_ℓ , the classical algorithm will infer it. It suffices to check whether the target is a Boolean function over \mathbf{x}_ℓ to determine whether the iteration ℓ will fail.

In order to detect whether the unknown target is a Boolean function over \mathbf{x}_ℓ , observe that a function cannot have two different outcomes on one input. When the target is a Boolean function over \mathbf{x}_ℓ , $\text{MEM}_\ell(u) \rightarrow \text{YES}$ if and only if $\overline{\text{MEM}}_\ell(u) \rightarrow \text{NO}$ for every $u \in \text{Val}_\ell$ (Lemma 3). Therefore, the unknown target is not a Boolean function over \mathbf{x}_ℓ if $\text{MEM}_\ell(u) \rightarrow \text{YES}$ and $\overline{\text{MEM}}_\ell(u) \rightarrow \text{YES}$ for some $u \in \text{Val}_\ell$. This simple observation motivates the following definition:

Definition 2. *A valuation $u \in \text{Val}_\ell$ ($1 \leq \ell$) is conflicting if $\text{MEM}_\ell(u) \rightarrow \text{YES}$ and $\overline{\text{MEM}}_\ell(u) \rightarrow \text{YES}$.*

The following lemma follows immediately from Lemma 3.

Lemma 4. *For any target Boolean function over a finite subset of \mathbf{x} , the target Boolean function is not over \mathbf{x}_ℓ if there is a conflicting valuation over \mathbf{x}_ℓ .*

Example (continued). Recall that \perp is a counterexample to both $\text{EQ}_1(\text{false})$ and $\text{EQ}_1(\text{true})$. By Lemma 3, $\text{MEM}_1(\perp) \rightarrow \text{YES}$ and $\overline{\text{MEM}}_1(\perp) \rightarrow \text{YES}$. Hence the unknown target is not a Boolean function over \mathbf{x}_1 .

Our first incremental learning algorithm is now clear. We parameterize the CDNF algorithm by the number of ordered variables. At iteration ℓ , we apply the parameterized CDNF algorithm and infer the unknown target as a Boolean function over \mathbf{x}_ℓ . If a

conflicting valuation is observed, we increment ℓ and move to the next iteration. Algorithm 3 shows the parameterized CDNF algorithm. Note that incremental equivalence queries are invoked in the parameterized algorithm. Similarly, incremental membership queries are used in the algorithm $walkTo(\ell, a, v)$ (Algorithm 2).

```

Input:  $\ell \in \mathbb{N} : 1 \leq \ell$ 
1  $t \leftarrow 0;$ 
2  $EQ_\ell(\text{true}) \rightarrow v;$ 
3 if  $v$  is YES then return true;
4  $t \leftarrow t + 1;$ 
5  $H_t, R_t, a_t \leftarrow \text{false}, \emptyset, v ;$  // assert  $\overline{MEM}_\ell(a_t) \rightarrow \text{YES}$ 
6  $EQ_\ell(\wedge_{i=1}^t H_i) \rightarrow v;$ 
7 if  $v$  is YES then return  $\wedge_{i=1}^t H_i$ ;
8  $I \leftarrow \{i : \llbracket H_i \rrbracket_\ell(v) = \perp\};$ 
9 if  $I = \emptyset$  then goto 4
10 foreach  $i \in I$  do
11    $r \leftarrow walkTo(\ell, a_i, v) ;$  // assert  $MEM_\ell(r) \rightarrow \text{YES}$ 
12   if  $a_i = r$  then raise conflict-found;
13    $R_i \leftarrow R_i \cup \{r\};$ 
14 end
15 foreach  $i = 1, \dots, t$  do  $H_i \leftarrow M_{\text{DNF}}(R_i \oplus a_i)(\mathbf{x}_\ell \oplus a_i);$ 
16 goto 6

```

Algorithm 3. $\wp\text{CDNF}(\ell)$

We give a parameterized generalization of $\Gamma(a, R)$ in Definition 3.

Definition 3. For $a \in Val_\ell$ ($1 \leq \ell$) and $R \subseteq Val_\ell$, define $\Delta_\ell(a, R)$ by

1. $\overline{MEM}_\ell(a) \rightarrow \text{YES};$
2. $MEM_\ell(r) \rightarrow \text{YES}$ for every $r \in R$;
3. $MEM_\ell(w) \rightarrow \text{NO}$ for every $r \in R$ and $w \in N_\ell(a, r)$.

The following lemma states that $\Delta_\ell(a_i, R_i)$ holds for $1 \leq i \leq t$ in the parameterized CDNF algorithm. Its proof is a generalization of those in Lemma 2. We call (a, R) a *speculative support with parameter ℓ* when $\Delta_\ell(a, R)$ holds.

Lemma 5. At line 6 of Algorithm 3 $\Delta_\ell(a_i, R_i)$ holds for every $1 \leq i \leq t$.

In order to decide conflicting valuations, recall that (a_i, R_i) 's are speculative supports with parameter ℓ . We have $\overline{MEM}_\ell(a_i) \rightarrow \text{YES}$ and $MEM_\ell(r) \rightarrow \text{YES}$ for every $r \in R_i$ (Lemma 5 and 1). If furthermore $a_i = r$, a_i is conflicting. By Lemma 4, the unknown target is not a Boolean function over \mathbf{x}_ℓ . We abort the parameterized algorithm by raising an exception (line 12, Algorithm 3).

Algorithm 4 gives our simple incremental learning algorithm. The CDNF+ algorithm starts from the variable ℓ equal to one. At iteration ℓ , it invokes the parameterized algorithm $\wp\text{CDNF}$ with parameter ℓ to infer the unknown target as a Boolean function over

```

1  $\ell \leftarrow 1;$ 
2 while  $\top$  do
3   try
4      $G = \wp\text{CDNF}(\ell)$ 
5     with conflict-found  $\implies \ell \leftarrow \ell + 1$ ;
6 end
7 return  $G$ ;

```

Algorithm 4. The CDNF+ Algorithm

\mathbf{x}_ℓ . If the parameterized algorithm infers the target, our simple algorithm terminates successfully. If the parameterized learning algorithm raises the exception *conflict-found*, the simple algorithm increments the variable ℓ and reiterates. The complexity of the CDNF+ algorithm follows from Theorem 1 and the number of iterations.

Theorem 2. Let f be an unknown target Boolean function over a finite subset of \mathbf{x} . The CDNF+ algorithm (Algorithm 4) infers f in $O(m^3|f|_{\text{CNF}}|f|_{\text{DNF}})$ incremental membership and $O(m|f|_{\text{CNF}}|f|_{\text{DNF}})$ incremental equivalence queries where m is the least number such that f is a Boolean function over \mathbf{x}_m .

The CDNF+ algorithm does not presume a fixed set of variables. It is hence applicable to scenarios where unknown targets are over indefinitely many variables. Moreover, the complexity of the CDNF+ algorithm depends on the number of ordered variables in the unknown target. If the target is a Boolean function over \mathbf{x}_1 , the CDNF+ algorithm will infer the target within $O(|f|_{\text{CNF}}|f|_{\text{DNF}})$ incremental membership queries. The classical learning algorithm in contrast needs $O(n^2|f|_{\text{CNF}}|f|_{\text{DNF}})$ membership queries if it infers the unknown target as a Boolean function over \mathbf{x}_n . The performance of the CDNF+ algorithm however depends on variable orderings and how incremental membership queries are resolved in practice. Section 5 evaluates these issues.

4.3 The CDNF++ Algorithm

We can actually do better than the CDNF+ algorithm. Observe that the simple incremental learning algorithm restarts the learning process at each iteration. All information from previous iterations known to the incremental algorithm is lost. The parameterized CDNF+ algorithm has to infer the unknown target from scratch. This is apparently not an economical management of information.

To retain the information obtained in previous iterations, we reuse parameterized speculative supports in each iteration. Each speculative support (a, R) with parameter ℓ satisfies the property $\Delta_\ell(a, R)$ at iteration ℓ (Lemma 5). We compute a speculative support (a^+, R^+) with parameter $\ell + 1$ from a speculative support (a, R) with parameter ℓ . After new parameterized speculative supports are constructed, we initiate the parameterized CDNF algorithm with the extended parameterized speculative supports and the conjecture derived from them. Information from previous iterations is thus retained.

Consider a speculative support (a, R) with parameter ℓ and a speculative support (a^+, R^+) with parameter $\ell + 1$. We have $a \in \text{Val}_\ell$ and $a^+ \in \text{Val}_{\ell+1}$. Similarly,

$R \subseteq Val_\ell$ and $R^+ \subseteq Val_{\ell+1}$. Each valuation in a speculative support with parameter ℓ is only short of the Boolean assignment to the variable $x_{\ell+1}$. To construct (a^+, R^+) from (a, R) , it suffices to extend the valuation a and every valuation over \mathbf{x}_ℓ in R by an assignment to $x_{\ell+1}$. To simplify the notation, we use the shorthand u^{+b} for $u[x_{\ell+1} \mapsto b]$ where $u \in Val_\ell$ and $b \in \mathbb{B}$. The following lemma follows from the definition.

Lemma 6. *Let $u \in Val_\ell$ ($1 \leq \ell$) be a valuation over \mathbf{x}_ℓ .*

1. *If $MEM_\ell(u) \rightarrow YES$, $MEM_{\ell+1}(u^{+\perp}) \rightarrow YES$ or $MEM_{\ell+1}(u^{+\top}) \rightarrow YES$.*
2. *If $MEM_\ell(u) \rightarrow NO$, $MEM_{\ell+1}(u^{+\perp}) \rightarrow NO$ and $MEM_{\ell+1}(u^{+\top}) \rightarrow NO$.*
3. *If $\overline{MEM}_\ell(u) \rightarrow YES$, $\overline{MEM}_{\ell+1}(u^{+\perp}) \rightarrow YES$ or $\overline{MEM}_{\ell+1}(u^{+\top}) \rightarrow YES$.*

Algorithm 5 expatiates the construction of (a^+, R^+) from (a, R) where $\Delta_\ell(a, R)$ holds. It starts by extending a . Recall that $\overline{MEM}_\ell(a) \rightarrow YES$. We can always find an extension a^+ with $\overline{MEM}_{\ell+1}(a^+)$ (Lemma 6). For the set $R^+ \subseteq Val_{\ell+1}$, the construction is not more difficult. We simply extend every valuation in R so that the extension yields YES on an incremental membership query.

```

Input:  $\ell \in \mathbb{N} : 1 \leq \ell; a \in Val_\ell : \overline{MEM}_\ell(a) \rightarrow YES; R \subseteq Val_\ell : MEM_\ell(r) \rightarrow YES$ 
        for every  $r \in R$ 
Output:  $a^+ \in Val_{\ell+1} : \overline{MEM}_{\ell+1}(a^+) \rightarrow YES; R^+ \subseteq Val_{\ell+1} :$ 
         $MEM_{\ell+1}(r^+) \rightarrow YES$  for every  $r^+ \in R^+$ 
        // assert  $\Delta_\ell(a, R)$ 
1  $b \leftarrow \text{if } \overline{MEM}_{\ell+1}(a^{+\perp}) \rightarrow YES \text{ then } \perp \text{ else } \top;$ 
2  $a^+ \leftarrow a^{+b};$ 
3  $R^+ \leftarrow \emptyset;$ 
4 foreach  $r \in R$  do
5    $c \leftarrow \text{if } MEM_{\ell+1}(r^{+b}) \rightarrow YES \text{ then } b \text{ else } \neg b;$ 
6    $R^+ \leftarrow R^+ \cup \{r^{+c}\};$ 
7 end
    // assert  $\Delta_{\ell+1}(a^+, R^+)$ 
8 return  $(a^+, R^+);$ 

```

Algorithm 5. $extendSupport(\ell, a, R)$

The following lemma states that the construction in Algorithm 5 is indeed correct. The only non-trivial part is to show that $N_{\ell+1}(a^+, r^+)$ consists of valuations yielding NO on incremental membership queries for every $r^+ \in R^+$.

Lemma 7. *Let $a \in Val_\ell$ ($1 \leq \ell$), $R \subseteq Val_\ell$, and $(a^+, R^+) = extendSupport(\ell, a, R)$ (Algorithm 5). If $\Delta_\ell(a, R)$, then $\Delta_{\ell+1}(a^+, R^+)$.*

With extended parameterized speculative supports, it is now straightforward to design our incremental learning algorithm (Algorithm 6). Similar to the simple incremental algorithm, the CDNF++ algorithm infers unknown target Boolean functions iteratively. At each iteration, it first proceeds as the parameterized CDNF algorithm. If the parameterized algorithm is able to infer the unknown target at iteration ℓ , our incremental algorithm terminates successfully and reports the result.

When the CDNF++ algorithm detects a conflicting valuation, it constructs extended parameterized speculative supports with Algorithm 5 (line 14, Algorithm 6). After extended parameterized speculative supports are obtained, the CDNF++ algorithm derives a new conjecture from them and enters the next iteration (line 19, Algorithm 6). The following theorem is proved by bounding the number of parameterized speculative supports and the size of R in each parameterized speculative support (a, R) .

```

1  $\ell \leftarrow 1;$ 
2  $t \leftarrow 0;$ 
3  $EQ_\ell(\text{true}) \rightarrow v;$ 
4 if  $v$  is YES then return true;
5  $t \leftarrow t + 1;$ 
6  $H_t, R_t, a_t \leftarrow \text{false}, \emptyset, v;$  // assert  $\overline{MEM}_\ell(a_t) \rightarrow \text{YES}$ 
7  $EQ_\ell(\wedge_{i=1}^t H_i) \rightarrow v;$ 
8 if  $v$  is YES then return  $\wedge_{i=1}^t H_i;$ 
9  $I \leftarrow \{i : \llbracket H_i \rrbracket_\ell(v) = \perp\};$ 
10 if  $I = \emptyset$  then goto 5
11 foreach  $i \in I$  do
12    $r \leftarrow \text{walkTo}(\ell, a_i, v);$  // assert  $MEM_\ell(r) \rightarrow \text{YES}$ 
13   if  $a_i = r$  then
14     foreach  $i = 1, \dots, t$  do  $(a_i, R_i) \leftarrow \text{extendSupport}(a_i, R_i);$ 
15      $\ell \leftarrow \ell + 1;$ 
16     goto 19
17    $R_i \leftarrow R_i \cup \{r\};$ 
18 end
19 foreach  $i = 1, \dots, t$  do  $H_i \leftarrow M_{\text{DNF}}(R_i \oplus a_i)(\mathbf{x}_\ell \oplus a_i);$ 
20 goto 7

```

Algorithm 6. The CDNF++ Algorithm

Theorem 3. Let f be an unknown target Boolean function over a finite subset of \mathbf{x} . The CDNF++ algorithm (Algorithm 6) infers f in $O(m^2|f|_{\text{CNF}}|f|_{\text{DNF}})$ incremental membership, $O(m|f|_{\text{CNF}})$ incremental non-membership, and $O(|f|_{\text{CNF}}|f|_{\text{DNF}})$ incremental equivalence queries where m is the least number that f is a Boolean function over \mathbf{x}_m .

Compared with the simple incremental learning algorithm, the CDNF++ algorithm improves linearly in the numbers of incremental membership and equivalence queries. In exchange, the sophisticated algorithm makes non-membership queries to extend parameterized speculative supports. Again, the performance of the CDNF++ algorithm depends on the order of variables and the efficiency of incremental query resolution. We give an assessment in the next section.

5 Experiments

We apply our incremental learning algorithms to the learning-based framework for loop invariant generation [11]. Let $\{ \delta \} \text{ while } \kappa \text{ do } S \{ \epsilon \}$ be an annotated loop with the *pre-condition* δ , the *post-condition* ϵ , and the *loop guard* κ . A *loop invariant* ι verifying the annotated loop is a quantifier-free formula such that $\delta \implies \iota$, $\iota \implies \epsilon \vee \kappa$, and $\iota \wedge \kappa \implies wp(S, \iota)$, where $wp(S, \iota)$ denotes the weakest precondition of ι for S .

The learning-based framework for loop invariant generation applies predicate abstraction [177] and adopts the CDNF algorithm [3] to infer the abstraction of a loop invariant for a given annotated loop. Using an SMT solver [615], a randomized mechanical teacher is devised to answer queries from the learning algorithm. Suppose n atomic predicates are used in the abstraction. Consider a membership query $MEM_n(v)$ with $v \in Val_n$. If the quantifier-free formula corresponding to the valuation v is stronger than δ , it must be stronger than any loop invariant ι for $\delta \implies \iota$. The mechanical teacher hence answers *YES* to the membership query $MEM_n(v)$. Similarly, if the corresponding formula of v is not stronger than $\epsilon \vee \kappa$, it is not included in any loop invariant ι for $\iota \implies \epsilon \vee \kappa$. The mechanical teacher thus answers *NO* to the membership query $MEM_n(v)$. In other cases, the mechanical teacher simply gives a random answer. Observe that random answers may yield different loop invariants in different runs. A multitude of loop invariants are exploited by the randomized teacher.

For predicate abstraction, atomic predicates are extracted from program texts heuristically [11]. If many irrelevant atomic predicates are extracted, the performance of classical learning will be impeded. We therefore apply incremental learning to improve the efficiency of the learning-based framework.

Two minor modifications derived from the domain knowledge are needed for this application. First, recall that any loop invariant must be stronger than the disjunction of the loop guard and the post-condition. An inferred loop invariant is likely to have atomic predicates from them. We hence start with these atomic predicates and infer loop invariants incrementally. This can be achieved by putting the atomic predicates of the loop guard and the post-condition in front of the variable set, and initializing the variable ℓ with the number of such predicates. Second, observe that random answers from the mechanical teacher may induce conflicting valuations. A conflict does not necessarily imply the lack of variables. To give the learning algorithm more chances to infer a loop invariant over the first ℓ atomic predicates, the variable ℓ is incremented only when the number of conflicts is greater than $\lceil \ell^{1.2} \rceil$. Otherwise, we restart the parameterized CDNF algorithm to infer a loop invariant over the first ℓ atomic predicates.

We compare the average performance of 1000 runs in five test cases. Data are collected from an Intel Core2 Quad Processor Q8200 running 64-bit Linux 2.6.32 with 4GB memory. Figure 11 shows our experimental results. Three learning algorithms (CDNF, CDNF+, and CDNF++) are compared in the same test cases from [11]. The number of atomic predicates is reported in the column “vars.” For the CDNF algorithm, it indicates the number of atomic predicates extracted from program texts. For the incremental learning algorithms, it indicates the average number of atomic predicates in a loop invariant. The column “cfcts” shows the average number of conflicting valuations induced by random answers or lack of variables. The columns “MEM”, “ $\overline{\text{MEM}}$ ”, and “EQ” are respectively the average numbers of membership, non-membership,

test case		vars	cflcts	MEM	MEM	EQ	MEM _{\$}	MEM _{\$}	EQ _{\$}	time
ide-ide-tape	CDNF	6.0	0.0	16.2	-	4.8	4.0	-	0.3	0.046s
	CDNF+	3.0	0.0	1.0	-	3.0	0.0	-	0.0	0.015s
	CDNF++	3.0	0.0	1.0	0.0	3.0	0.0	0.0	0.0	0.015s
ide-wait-ireason	CDNF	8.0	1.6	85.5	-	32.9	14.9	-	7.8	0.237s
	CDNF+	4.0	0.0	8.0	-	9.5	1.0	-	0.0	0.044s
	CDNF++	4.0	0.0	19.0	0.0	29.0	0.0	0.0	0.0	0.088s
parser	CDNF	20.0	20.5	10203.9	-	1286.9	1306.6	-	44.9	41.044s
	CDNF+	9.0	0.0	97.3	-	32.4	36.8	-	0.0	0.501s
	CDNF++	9.0	0.0	304.8	0.0	91.0	8.5	0.0	0.0	1.006s
usb-message	CDNF	10.0	0.0	21.1	-	6.8	1.0	-	0.0	0.097s
	CDNF+	5.0	0.0	19.5	-	6.6	2.2	-	0.0	0.065s
	CDNF++	5.0	0.0	60.9	0.0	21.7	9.6	0.0	0.0	0.147s
vpr	CDNF	7.0	0.9	4.6	-	6.4	20.1	-	3.4	0.070s
	CDNF+	5.1	0.8	4.0	-	5.9	17.7	-	3.0	0.057s
	CDNF++	5.0	0.1	5.6	3.0	9.2	21.9	0.0	2.0	0.064s

Fig. 1. Experimental Results – Heuristic Variable Ordering

and equivalence queries answered conclusively. The columns “MEM_{\$}”, “MEM̄_{\$}”, and “EQ_{\$}” show the average numbers of random membership, non-membership, and equivalence queries respectively. The column “time” indicates average running time.

With our simple heuristic variable ordering, the CDNF+ algorithms performs better than the classical learning algorithm in all test cases. The more sophisticated CDNF++ algorithm is outperformed by the classical algorithm in only one test case (usb-message). Both incremental learning algorithms improve the most complicated case `parser` significantly. The classical learning algorithm takes about 41 seconds to infer a loop invariant in this test case. The CDNF+ and CDNF++ algorithms use about .5 and 1 second respectively in the same test case. Across the five test cases, the CDNF+ and CDNF++ algorithms have expected speedups of 59.8% and 36.9% respectively.

We now evaluate the worst-case performance of the incremental learning algorithms. To this end, we randomly order the set of atomic predicates extracted from program texts at each run. Starting from the first variable in a random variable ordering, our incremental learning algorithms are invoked to infer loop invariants. Similarly, we invoke the classical CDNF algorithm on all randomly ordered variables at each run. We compare the average of 1000 runs in each test case. Figure 2 gives the results.

With random variable orderings, the incremental learning algorithms perform comparably to the classical learning algorithm in all test cases but `usb-message`. For this particular case, conflicts are negligible when all atomic predicates are used. Incremental learning, on the other hand, needs to enlarge the set of atomic predicates 8 times. Subsequently, both incremental learning algorithms make lots of useless queries before a loop invariant is inferred. Also note that the CDNF algorithm performs significantly better with random variable orderings in the test case `parser`. Yet the classical algorithm still requires about 12 seconds to infer a loop invariant. In comparison, our incremental algorithms are an order of magnitude faster with our heuristic variable ordering (cf Figure 1). Using random variable orderings, we observe 19.4% and 18.5% of

test case		vars	cflcts	MEM	MEM	EQ	MEM _{\$}	MEM _{\$}	EQ _{\$}	time
ide-ide-tape	CDNF	6.0	0.1	13.0	-	5.0	3.6	-	0.4	0.048s
	CDNF+	2.7	2.5	4.1	-	10.5	0.9	-	0.0	0.028s
	CDNF++	2.8	2.7	5.2	0.0	13.2	1.6	0.0	0.1	0.037s
ide-wait-ireason	CDNF	8.0	1.6	87.8	-	32.0	14.2	-	7.6	0.247s
	CDNF+	6.9	7.6	76.4	-	51.7	12.6	-	5.1	0.236s
	CDNF++	6.8	7.4	83.0	3.4	56.0	17.5	0.4	4.5	0.256s
parser	CDNF	20.0	5.6	2948.4	-	405.6	563.7	-	12.6	11.961s
	CDNF+	19.0	31.1	4343.5	-	942.0	783.0	-	8.9	18.143s
	CDNF++	19.1	31.5	3365.1	19.3	572.8	757.1	0.4	9.1	13.504s
usb-message	CDNF	10.0	0.0	21.4	-	7.3	1.0	-	0.0	0.094s
	CDNF+	8.1	8.1	47.2	-	44.1	3.1	-	0.0	0.205s
	CDNF++	8.4	8.4	39.8	3.5	35.1	5.0	0.0	0.0	0.181s
vpr	CDNF	7.0	1.6	9.5	-	9.4	33.0	-	6.3	0.112s
	CDNF+	4.4	4.4	7.3	-	16.4	16.2	-	6.4	0.082s
	CDNF++	5.1	5.6	15.9	1.4	22.5	24.0	1.0	6.5	0.119s

Fig. 2. Experimental Results – Random Variable Orderings

slowdowns respectively from the CDNF+ and CDNF++ algorithms across the five test cases. Note that the test case `usb-message` alone registers a slowdown of more than 90%. The incremental learning algorithms in fact perform slightly better than the classical algorithm for the other four test cases on average (5.3% for CDNF+ and 0.1% for CDNF++). Also recall that this is the worst-case scenario for incremental learning. As in loop invariant inference, heuristics for choosing sensible variable orderings are often available for most applications. Our incremental learning algorithms should outperform the classical algorithm with the domain knowledge in practice.

6 Conclusion

Classical learning algorithms for Boolean functions assume a fixed number of variables for unknown targets. The assumption precludes applications where indefinitely many variables are needed. It can also be unexpectedly inefficient at the presence of irrelevant variables. We address the problem by inferring unknown targets through enlarging numbers of ordered variables. Our experiments show that incremental learning attains significant improvement with a simple heuristic variable ordering. They also suggest manageable slowdowns in the worst-case scenario with random variable orderings.

Applications of incremental learning in formal verification are under investigation. Particularly, problems in program verification inherently have indefinitely many variables in unknown targets. Applying incremental learning to program verification will be interesting. We are working on applications in automated assume-guarantee reasoning. Domain knowledge about contextual assumptions will be essential in this application.

Acknowledgement. We thank the invaluable comments from anonymous referees.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* 75(2), 87–106 (1987)
2. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated Assume-Guarantee Reasoning by Abstraction Refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
3. Bshouty, N.H.: Exact learning Boolean function via the monotone theory. *Information and Computation* 123(1), 146–153 (1995)
4. Chen, Y.-F., Clarke, E.M., Farzan, A., He, F., Tsai, M.-H., Tsay, Y.-K., Wang, B.-Y., Zhu, L.: Comparing Learning Algorithms in Automated Assume-Guarantee Reasoning. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part I. LNCS, vol. 6415, pp. 643–657. Springer, Heidelberg (2010)
5. Chen, Y.-F., Clarke, E.M., Farzan, A., Tsai, M.-H., Tsay, Y.-K., Wang, B.-Y.: Automated Assume-Guarantee Reasoning through Implicit Learning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 511–526. Springer, Heidelberg (2010)
6. Dutertre, B., Moura, L.D.: The Yices SMT solver. Technical report, SRI International (2006)
7. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202. ACM (2002)
8. Gheorghiu, M., Giannakopoulou, D., Păsăreanu, C.S.: Refining Interface Alphabets for Compositional Verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
9. Giannakopoulou, D., Păsăreanu, C.S.: Special issue on learning techniques for compositional reasoning. *Formal Methods in System Design* 32(3), 173–174 (2008)
10. Howar, F., Steffen, B., Merten, M.: Automata Learning with Automated Alphabet Abstraction Refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011)
11. Jung, Y., Kong, S., Wang, B.-Y., Yi, K.: Deriving Invariants by Algorithmic Learning, Decision Procedures, and Predicate Abstraction. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 180–196. Springer, Heidelberg (2010)
12. Jung, Y., Lee, W., Wang, B.-Y., Yi, K.: Predicate Generation for Learning-Based Quantifier-Free Loop Invariant Inference. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 205–219. Springer, Heidelberg (2011)
13. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press (1994)
14. 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)
15. Kroening, D., Strichman, O.: Decision Procedures - an algorithmic point of view. EATCS. Springer (2008)
16. Lee, W., Wang, B.Y., Yi, K.: Termination Analysis with Algorithmic Learning. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 88–104. Springer, Heidelberg (2012)
17. Saídi, H., Graf, S.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

Interpolants as Classifiers^{*}

Rahul Sharma¹, Aditya V. Nori², and Alex Aiken¹

¹ Stanford University

{sharmar,aiken}@stanford.edu

² Microsoft Research India

adityan@microsoft.com

Abstract. We show how interpolants can be viewed as classifiers in supervised machine learning. This view has several advantages: First, we are able to use off-the-shelf classification techniques, in particular support vector machines (SVMs), for interpolation. Second, we show that SVMs can find relevant predicates for a number of benchmarks. Since classification algorithms are predictive, the interpolants computed via classification are likely to be invariants. Finally, the machine learning view also enables us to handle superficial non-linearities. Even if the underlying problem structure is linear, the symbolic constraints can give an impression that we are solving a non-linear problem. Since learning algorithms try to mine the underlying structure directly, we can discover the linear structure for such problems. We demonstrate the feasibility of our approach via experiments over benchmarks from various papers on program verification.

Keywords: Static analysis, interpolants, machine learning.

1 Introduction

Problems in program verification can be formalized as learning problems. In particular, we show how interpolants [4][17][11] that are useful heuristics for computing “simple” proofs in program verification can be looked upon as classifiers in supervised machine learning. Informally, an interpolant is a predicate that separates good or positive program states from bad or negative program states and a set of appropriately chosen interpolants forms a program proof. Our main technical insight is to view interpolants as classifiers that distinguish positive examples from negative examples. This view allows us to make the following contributions:

- We are able to use state-of-the-art classification algorithms for the purpose of computing interpolants. Specifically, we show how to use support vector machines (SVMs) [21] for binary classification to compute interpolants.

* This work was supported by NSF grant CCF-0915766 and the Army High Performance Computing Research Center.

- Since classification algorithms are predictive, the interpolants we compute are relevant predicates for program proofs. We show that we can discover inductive invariants for a number of benchmarks. Moreover, since SVMs are routinely used in large scale data processing, we believe that our approach can scale to verification of practical systems.
- Classification based interpolation also has the ability to detect superficial non-linearities. As shown in Section 4, even if the underlying problem structure is linear, the symbolic constraints can give an impression that we are solving a non-linear problem. Since our algorithm mines the underlying structure directly, we can discover the linear structure for such problems.

The rest of the paper is organized as follows. We informally introduce our technique by way of an example in Section 1. In Section 2, we describe necessary background material including a primer on SVMs. Section 3 describes the main results of our work. We first introduce a simple algorithm BASIC that uses an SVM as a black box to compute a candidate interpolant and we formally characterize its output. SVMs rely on the assumption that the input is linearly separable. Hence, we give an algorithm SVM-I (which makes multiple queries to an SVM) that does not rely on the linear separability assumption and prove correctness of SVM-I. We augment BASIC with a call to SVM-I; the output of the resulting algorithm is still not guaranteed to be an interpolant. This algorithm fails to output an interpolant when we do not have a sufficient number of positive and negative examples. Finally, we describe an algorithm INTERPOLANT that generates a sufficient number of positive and negative examples by calling BASIC iteratively. The output of INTERPOLANT is guaranteed to be an interpolant and we formally prove its soundness. In Section 4, we show how our technique can handle superficial non-linearities via an example that previous techniques are not capable of handling. Section 5 describes our implementation and experiments over a number of benchmarks. Section 6 places our work in the context of existing work on interpolants and machine learning. Finally, Section 7 concludes with some directions for future work.

1.1 An Overview of the Technique

We show an example of how our technique for interpolation discovers invariants for program verification. Consider the program in Fig. 1. This program executes the loop at line 2 a non-deterministic number of times. Upon exiting this loop, the program decrements x and y until x becomes zero. At line 6, if y is not 0 then we go to an error state. To prove that the `error()` statement is unreachable, we need invariants for the loops. We follow the standard verification by interpolants recipe and try to find invariants by finding interpolants for finite infeasible traces of the program. The hope is that the interpolants thus obtained will give us predicates that generalize well. In particular, we aim to obtain an inductive loop invariant. For example, $x = y$ is a sufficiently strong loop invariant for proving the correctness of Fig. 1.

Suppose we consider a trace that goes through all the loops once. Then we get the following infeasible trace: (1, 2, 3, 2, 4, 5, 4, 6, 7). We decompose this

```

foo( )
{
1: x = y = 0;
2: while (*)
3:   { x++; y++; }
4: while ( x != 0 )
5:   { x--; y--; }
6: if ( y != 0 )
7:   error();
}

```

Fig. 1. Motivating example for our technique

trace into two parts A and B and thereby find interpolants for this infeasible trace. A represents the values of x and y obtained after executing lines 1, 2, and 3. B represents the values of x and y such that if we were to execute lines 4, 5, 6, and 7 then the program reaches the `error()` statement. Now, we have (A, B) where $A \wedge B \equiv \perp$:

$$\begin{aligned} A &\equiv x_1 = 0 \wedge y_1 = 0 \wedge \text{ite}(b, x = x_1 \wedge y = y_1, x = x_1 + 1 \wedge y = y_1 + 1) \\ B &\equiv \text{ite}(x = 0, x_2 = x \wedge y_2 = y, x_2 = x - 1 \wedge y_2 = y - 1) \wedge x_2 = 0 \wedge \neg(y_2 = 0) \end{aligned}$$

Here *ite* stands for if-then-else. As is evident from this example, A is typically the set of reachable states and B is the set of states that reach `error()`. An interpolant is a proof that shows A and B are disjoint and is expressed using the common variables of A and B . In this example, x and y are the variables common to A and B . Our technique for finding an interpolant between A and B operates as follows: First, we compute samples of values for (x, y) that satisfy the predicates A and B . Fig. 2 plots satisfying assignments of A as '+'s (points $(0, 0)$ and $(1, 1)$) and of B as 'o's (points $(1, 0)$ and $(0, 1)$). Next, we use an SVM to find lines separating the 'o's from the '+'s.

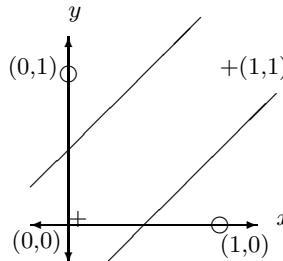


Fig. 2. Finding interpolants using an SVM

We consider the 'o' points one by one and ask an SVM to find a line which separates the chosen 'o' point from the '+'s. On considering $(0, 1)$, we get the line $2y = 2x + 1$ and from $(1, 0)$ we obtain $2y = 2x - 1$. Using these two lines,

we obtain the interpolant, $2y \leq 2x + 1 \wedge 2y \geq 2x - 1$. It can be checked that this predicate is an invariant and is sufficient to prove the `error()` statement of Fig. 11 unreachable.

We will see in Section 2.1 that we can easily obtain the stronger predicate $x = y$. Intuitively, we just have to translate the separating lines as close to the +'s as possible while ensuring that they still separate the +'s from the o's.

2 Preliminaries

Let A and B be two formulas in the theory of *linear arithmetic*:

$$\phi ::= w^T x + d \geq 0 \mid \text{true} \mid \text{false} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi$$

$w = (w_1, \dots, w_n)^T \in \mathbb{R}^n$ is a *point*: an n -dimensional vector of constants; $x = (x_1, \dots, x_n)^T$ is an n -dimensional vector of variables. The *inner product* $\langle w, x \rangle$ of w and x is $w^T x = w_1 x_1 + \dots + w_n x_n$. The equation $w^T x + d = 0$ is a *hyperplane* in $n-1$ dimensions. Each hyperplane corresponds to two *half-spaces*: $w^T x + d \geq 0$ and $w^T x + d \leq 0$. A half-space divides \mathbb{R}^n into two parts: variable values that satisfy the half-space and those which do not. For example, $x - y = 0$ is a 1-dimensional hyperplane, $x - y + 2z = 0$ is a 2-dimensional hyperplane, and $x \geq y$ and $x \leq y$ are half-spaces corresponding to the hyperplane $x = y$.

Suppose $A \wedge B \equiv \perp$, i.e., there is no assignment to variables present in the formula $A \wedge B$ that makes the formula *true*. Informally, an *interpolant* is a simple explanation as to why A and B are disjoint. Formally, it is defined as follows:

Definition 1 (Interpolant [17]). An *interpolant* for a pair of formulas (A, B) such that $A \wedge B \equiv \perp$ is a formula I satisfying $A \Rightarrow I$, $I \wedge B \equiv \perp$, and I refers only to variables common to both A and B .

Let $\text{Vars}(A, B)$ denote the common variables of A and B . We refer to the values assigned to $\text{Vars}(A, B)$ by satisfying assignments of A as *positive examples*. Dually, *negative examples* are values assigned to $\text{Vars}(A, B)$ by satisfying assignments of B . *Sampling* is the process of obtaining positive and negative examples given A and B . For instance, sampling from $(A \equiv y < x)$ and $(B \equiv y > x)$ with common variables x and y , can give us a positive example $(1, 0)$ and a negative example $(0, 1)$.

A well studied problem in machine learning is *binary classification*. The input to the binary classification problem is a set of points with associated *labels*. By convention, these labels are $l \in \{+1, -1\}$. The goal of the binary classification problem given points with labels is to find a *classifier* $C : \text{point} \rightarrow \{\text{true}, \text{false}\}$ s.t. $C(a) = \text{true}$ for all points a with label $+1$, and $C(b) = \text{false}$ for all points b with label -1 . This process is called *training* a classifier and the set of labeled points is called the *training data*. The goal is to find classifiers that are predictive, i.e., even if we are given a new labeled point w with label l not contained in the training data then it should be very likely that $C(w)$ is *true* iff $l = +1$.

Our goal in this paper is to apply standard binary classification algorithms to positive and negative examples to obtain interpolants. We will assign positive

examples the label +1 and the negative examples the label -1 to obtain the training data. We are interested in classifiers, in the theory of linear arithmetic, that *classify correctly*.

Definition 2 (Correct Classification). A classifier C classifies correctly on training data X if for all positive examples $a \in X$, $C(a) = \text{true}$, and for all negative examples $b \in X$, $C(b) = \text{false}$. If there exists a positive example a such that $C(a) = \text{false}$ (or a negative example b such that $C(b) = \text{true}$), then C is said to have misclassified a (or b).

There are classification algorithms that need not classify correctly on training data [10]. These are useful because typically the data in machine learning is noisy. A classifier that misclassifies a training example is definitely not an interpolant. Hence we focus on classifiers that classify correctly on training data. In particular, we use optimal margin classifiers generated by SVMs.

2.1 SVM Primer

We provide some basic background on SVMs in the context of binary classification using half-spaces. Let us denote the training data by X , the set of positive examples by X^+ , and the set of negative examples by X^- .

Let us assume that the training data X is *linearly separable*: there exists a hyperplane, called a *separating hyperplane*, $w^T x + d = 0$ such that $\forall a \in X^+, w^T a + d > 0$ and $\forall b \in X^-, w^T b + d < 0$. For linearly separable training data, an SVM is guaranteed to terminate with a separating hyperplane. To use a separating hyperplane to predict the label of a new point z we simply compute $\text{sgn}(w^T z + d)$. In other words, if $w^T z + d \geq 0$ then we predict the label to be +1 and -1 otherwise.

An interesting question to consider is the following: If there are multiple separating hyperplanes then which one is the best? If a point is away from the separating hyperplane, say $w^T x + d \gg 0$, then our prediction that x is a positive example is reasonably confident. On the other hand, if x is very close to the separating hyperplane then our prediction is no longer confident as a minor perturbation can change the predicted label. We say such points have a very low *margin*. The *optimal margin classifier* is the separating hyperplane that maximizes the distance from the points nearest to it. The points closest to the optimal margin classifier are called *support vectors*. An SVM finds the optimal margin classifier and the support vectors given linearly separable training data efficiently [21] by solving a convex optimization problem.

An example of SVM in action is shown in Fig. 3. The positive examples are shown by +'s and negative examples by o's. Line 4 is a separating hyperplane and we can observe that several points of training data lie very close to it and hence its predictions are not so confident. Line 2 is the optimal margin classifier. The points on the dotted lines are closest to the optimal margin classifier and hence are the support vectors.

We observe that using SVMs provides us with a choice of half-spaces for the classifier. We can return the half-space above line 2 as a classifier. All positive

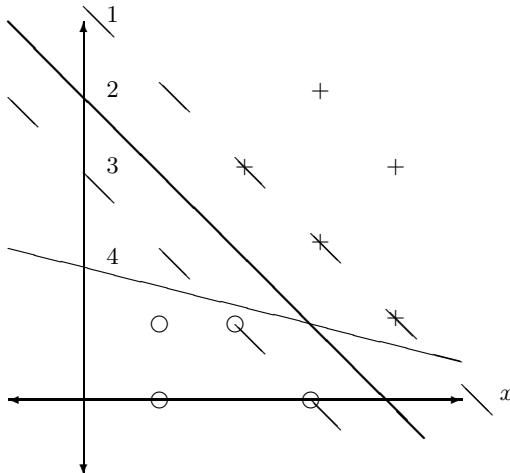


Fig. 3. Line 2 and line 4 are separating hyperplanes. The support vectors for optimal margin classifier (line 2) lie on dotted lines.

examples are contained in it and all negative examples are outside it. Or we can return the half-space above line 1 and that will be a stronger predicate. Or we can return the negation of the half-space below line 3 and that will be a weaker predicate. Or any line parallel to line 2 and lying between line 1 and line 3 will work. The choice of predicate depends on the application (i.e., the program verification tool that consumes these predicates) and all these predicates can be easily generated by taking a linear combinations of the support vectors.

3 Classification Based Algorithms for Interpolation

We now discuss an algorithm for computing interpolants using an SVM as a black box. We start with a basic version as described in Fig. 4. BASIC takes as input two predicates A and B over the theory of linear arithmetic and generates as output a half-space h over the common variables of A and B . BASIC also has access to (possibly empty) sets of already known positive examples X^+ and negative examples X^- .

```

BASIC( $A, B$ )
  Vars := Common variables of  $A$  and  $B$ 
  Add Samples( $A, X^+$ ) to  $X^+$ 
  Add Samples( $B, X^-$ ) to  $X^-$ 
  SV := SVM( $X^+, X^-$ )
   $h := \text{Process}(SV, X^+, X^-);$ 
  return  $h$ 

```

Fig. 4. The basic algorithm for computing a separating hyperplane

BASIC first computes the variables common to both A and B and stores them in the set $Vars$. It then computes the positive examples X^+ by repeatedly asking a theorem prover for satisfying assignments of A not already present in X^+ (by calling $Samples(A, X^+)$). The values assigned to variables in $Vars$ by these satisfying assignments are stored in X^+ . The negative examples X^- are computed from B in a similar fashion (by calling $Samples(B, X^-)$). Let us assume that X^+ and X^- are linearly separable. Next, we compute the support vectors (SV of Fig. 4) for X^+ and X^- by calling an off-the-shelf SVM to generate the optimal margin classifier. The result is then processed via the call to procedure *Process* which takes a linear combination of support vectors in SV to obtain the classifier $h = w^T x + d \geq 0$ s.t. $w^T x + d = 0$ is the optimal margin classifier between X^+ and X^- , and $\forall a \in X^+. h(a) > 0$ and $\forall b \in X^-. h(b) < 0$. This half-space h is returned as output after correction for minor numerical artifacts (say rounding 4.9996 to 5). *Process* can be modified to produce stronger or weaker predicates (Section 2.1). The output of BASIC is characterized by the following lemma:

Lemma 1 (Correctness of SVM). *Given positive examples X^+ which are linearly separable from negative examples X^- , SVM and Process compute a half-space h s.t. $\forall a \in X^+. h(a) > 0$ and $\forall a \in X^-. h(a) < 0$.*

Proof. The lemma follows from the fact that SVM returns an optimal margin classifier under the assumption that X^+ and X^- are linearly separable, and that rounding performed by *Process* does not affect the predicted label of any example in X^+ or X^- .

However, the algorithm BASIC has two major problems:

1. SVM will produce a sound output only when X^+ and X^- are linearly separable.
2. BASIC computes a separator for X^+ and X^- which might or might not separate all possible models of A from all possible models of B .

We will now provide partial solutions for both of these concerns.

3.1 Algorithm for Intersection of Half-spaces

Suppose BASIC samples X^+ and X^- which are not linearly separable. If we denote x_1, \dots, x_n as the variables contained in $Vars$ then there is an obvious (albeit not very useful) separator between X^+ and X^- given by the following predicate:

$$P = \bigvee_{(a_1, \dots, a_n) \in X^+} x_1 = a_1 \wedge \dots \wedge x_n = a_n$$

Observe that $\forall a \in X^+. P(a) = true$ and $\forall b \in X^-. P(b) = false$. The predicate P is a union (or disjunction) of intersection (or conjunction) of half-spaces. To avoid the discovery of such specific predicates, we restrict ourselves to the case where the classifier is either a union or an intersection of half-spaces. This means that we will not be able to find classifiers in all cases even if they exist in the theory of linear arithmetic. We will now give an algorithm which is only guaranteed

to succeed if there exists a classifier which is an intersection of half-spaces. We only discuss the case of intersection here as finding union of half-spaces can be reduced to finding intersection of half-spaces by solving the dual problem.

Definition 3 (Problem Statement). *Given X^+ and X^- such that there exist a set of half-spaces $H = \{h_1, \dots, h_n\}$ classifying X^+ and X^- correctly (i.e., $\forall a \in X^+. \bigwedge_{i=1}^n h_i(a)$ and $\forall b \in X^-. \neg \bigwedge_{i=1}^n h_i(b)$) find H .*

```

SVM-I( $X^+, X^-$ )
 $H := \text{true}$ 
 $Misclassified := X^-$ 
while  $|Misclassified| \neq 0$ 
    Arbitrarily choose  $b$  from  $Misclassified$ 
     $h := \text{Process}(SVM(X^+, \{b\}), X^+, X^-)$ 
     $\forall b' \in Misclassified \text{ s.t. } h(b') < 0 : \text{remove } b' \text{ from } Misclassified$ 
     $H := H \wedge h$ 
end while
return  $H$ 

```

Fig. 5. Algorithm for classifying by intersection of half-spaces

We find such a classifier using the algorithm of Fig. 5. We initialize the classifier H to $true$ or $0 \leq 0$. Next we compute the set of examples misclassified by H . $\forall a \in X^+. H(a) = true$ and hence all positive examples have been classified correctly. $\forall b \in X^-. H(b) = true$ and hence all negative examples have been misclassified. Therefore we initialize the set of misclassified points, $Misclassified$, by X^- . We consider a misclassified element b and find the support vectors between b and X^+ . Using the assumption that a classifier using intersection of half-spaces exists for X^+ and X^- , we can show that b is linearly separable from X^+ . Using Lemma II, we will obtain a half-space $h = w^T x + d \geq 0$ for which $h(b) < 0$. We will add h to our classifier and remove the points which h classifies correctly from the set of misclassified points. In particular, b is no longer misclassified and we repeat until all examples have been classified correctly. A formal proof of the following theorem can be developed along the lines of the argument above:

Theorem 1 (Correctness of SVM-I). *If there exists an intersection of half-spaces, H , that can correctly classify X^+ and X^- then SVM-I is a sound and complete procedure for finding H .*

We make the following observations about SVM-I:

- The classifier found depends on the order by which the misclassified element b is chosen and different choices can lead to different classifiers.
- In the worst case, it is possible that SVM-I will find as many half-spaces as the number of negative examples. But since optimal margin classifiers generalize well, the worst case behavior does not usually happen in practice.

- SVM-I is related to the problem of “learning intersection of half-spaces”. In the latter problem, given positive and negative examples, the goal of the learner is to output an intersection of half-spaces which classifies any new example correctly with high probability. There are several negative results about learning intersection of half-spaces. If no assumptions are made regarding the distribution from which examples come from, we cannot learn intersection of even 2 half-spaces in polynomial time unless RP=NP [2013].

SVM-I can be incorporated into BASIC by replacing the calls to *SVM* and *Process* with SVM-I in Fig. 4. Now BASIC with SVM-I can find classifiers when X^+ and X^- are not linearly separable but can be separated by an intersection of half-spaces.

3.2 A Sound Algorithm

We observe that BASIC, with or without SVM-I, only finds classifiers between X^+ and X^- . The way BASIC is defined, these candidate interpolants are over the common variables of A and B . But if we do not have enough positive and negative examples then a classifier between X^+ and X^- is not necessarily an interpolant. When this happens, we need to add more positive and negative examples refuting the candidate interpolant.

```

INTERPOLANT( $A, B$ )
 $X^+, X^- := \emptyset$ 
while true
     $H := \text{BASIC}(A, B)$  // BASIC with SVM-I
    if  $SAT(A \wedge \neg H)$ 
        Add satisfying assignment to  $X^+$  and continue
    if  $SAT(B \wedge H)$ 
        Add satisfying assignment to  $X^-$  and continue
    break
return  $H$ 
```

Fig. 6. A sound algorithm for interpolation

The algorithm INTERPOLANT computes a classifier H which classifies X^+ and X^- correctly i.e., $\forall a \in X^+. H(a) = \text{true}$ and $\forall b \in X^-. H(b) = \text{false}$ by calling BASIC with SVM-I. If H is implied by A and is unsatisfiable in conjunction with B then we have found an interpolant and we exit the loop. Otherwise we update X^+ and X^- and try again. We have the following theorem:

Theorem 2 (Soundness of INTERPOLANT). $\text{INTERPOLANT}(A, B)$ terminates if and only if the output H is an interpolant between A and B .

Proof. The output H is defined over the common variables of A and B (follows from the output of BASIC).

- only if** : Let $\text{INTERPOLANT}(A, B)$ terminate. This means that both conditions $B \wedge H \equiv \perp$ and $A \wedge \neg H \equiv \perp$ must be satisfied (these are conditions for reaching `break` statement), which in turn implies that $A \Rightarrow H$ holds and therefore H is an interpolant of A and B .
- if** : Let H be an interpolant of A and B . This means that $A \Rightarrow H$ and hence $A \wedge \neg H \equiv \perp$. $B \wedge H \equiv \perp$ holds because H is an interpolant and therefore, the `break` statement is reachable and $\text{INTERPOLANT}(A, B)$ terminates.

4 Handling Superficial Non-linearities

Most program verification engines do not reason about non-linear arithmetic directly. They try to over-approximate non-linear functions, say by using uninterpreted function symbols. In this section, we discuss how to use our technique to over-approximate non-linear arithmetic by linear functions.

Suppose $A \wedge B \equiv \perp$ and A is a non-linear predicate. If we can find a linear interpolant I between A and B then $A \Rightarrow I$. Hence I is a linear over-approximation of the non-linear predicate A . We discuss, using an example, how such a predicate I can be useful for program verification.

Suppose we want to prove that line 5 is unreachable in Fig. 7. There are some lines which are commented. These will be considered later. This program assigns z non-deterministically and does some non-linear computations. If we can show that an over-approximation of reachable states after line 3 is disjoint from $x = 2 \wedge y \neq 2$ then have a proof that `error()` is unreachable.

```

foo()
{
    // do{
    1:    z = nondet();
    2:    x = 4 * sin(z) * sin(z);
    3:    y = 4 * cos(z) * cos(z);
    // } while (*);
    4:    if ( x == 2 && y != 2 )
    5:        error() ;
}

```

Fig. 7. A contrived example with superficial non-linearities

We use our technique for computing interpolants over the non-linear predicates to construct an easy to analyze over-approximation of this program. We want to find an interpolant of the following predicates (corresponding to the infeasible trace (1, 2, 3, 4, 5)):

$$\begin{aligned} A &\equiv x = 4\sin^2(z) \wedge y = 4\cos^2(z) \\ B &\equiv x = 2 \wedge y \neq 2 \end{aligned}$$

Observe that SVMs consume examples and are agnostic to how the examples are obtained. Since A is non-linear, we can obtain positive examples by randomly

substituting values for z in A and recording the values of x and y . Since B is linear, we can ask an SMT solver [19] for satisfying assignments of B to obtain the negative examples. We have plotted one possible situation in Fig. 8 – the positive and negative examples are represented by +'s and o's respectively. Running SVM-I and choosing the stronger predicate from the available choices (Section 2.1) generates the predicate $P \equiv (x + y = 4)$. We remark that to obtain this predicate, we only need one negative example above, one negative example below, one positive example to the left, and one positive example to the right of $(2, 2)$. Adding more examples will leave P unaffected, due to the way optimal margin classifier is defined (Section 2.1). This shows the robustness of the classifier. That is, once a sufficient number of samples have been obtained then the classifier is not easily perturbed by changes in the training data.

Now we need to verify that P is actually an interpolant. We use an SMT solver to show that $P \wedge B \equiv \perp$. To show $A \Rightarrow P$ can be hard. For this example, any theorem prover with access to the axiom $\sin^2(x) + \cos^2(x) = 1$ will succeed. But we would like to warn the reader that the verification step, where we check $A \Rightarrow I$ and $I \wedge B \equiv \perp$, can become intractable for arbitrary non-linear formulas.

Using the interpolant P , we can replace Fig. 7 by its over-approximation given in Fig. 9 for verification. A predicate abstraction engine using predicates $\{x + y = 4, x = 2, y = 2\}$ can easily show the correctness of the program of Fig. 9. Moreover, suppose we uncomment the lines which have been commented out in Fig. 7. To verify the resulting program we need a sufficiently strong loop invariant. To find it we consider a trace executing the loop once and try to find the interpolant. We do the exact same analysis we did above and obtain the interpolant $(x + y = 4)$. This predicate is an invariant and is sufficient to prove the unreachability of `error()`.

Other techniques for interpolation fail on this example because either they replace \sin and \cos by uninterpreted functions [13][24] or because of the restricted expressivity of the range of interpolants computed (e.g. combination of boxes [15]). We succeed on this example because of two reasons:

1. We are working with examples and hence we are not over-approximating the original constraints.
2. SVM succeeds in computing a predicate which generalizes well.

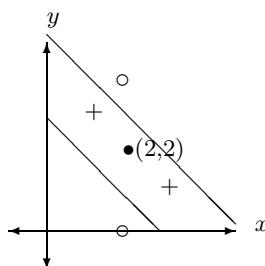


Fig. 8. Positive and negative examples for Fig. 7. The lines show classifiers.

```

foo()
{
    assume ( x + y == 4 );
    if ( x == 2 && y != 2)
        error();
}

```

Fig. 9. An over-approximation of Fig. 7

5 Experiments

We have implemented a prototype version of the algorithm described in this paper in 1000 lines of C++ using LIBSVM [3] for SVM queries and the Z3 theorem prover [19]. Specifically, we use the C-SVC algorithm with a linear kernel for finding the optimal margin classifier. C-SVC is parametrized by a cost parameter c . A low value of c allows the generated classifier to make errors on the training data. Since we are interested in classifiers that classify correctly, we assign a very high value to c (1000 in our experiments). The input to our implementation is two SMT-LIB formulas and the output is also obtained as an SMT-LIB formula. We try to sample for at most ten distinct positive and negative examples each before BASIC makes a call to LIBSVM. In these experiments, the classifier is described by the hyperplane parallel to the optimal margin classifier and passing through the positive support vectors. We consider the half-space, corresponding to this hyperplane, such that the negative examples lie outside the half-space. Hence we are considering the strongest predicate from the options provided to us by SVM (Section 2.1).

We have tried our technique on small programs and our results are quite encouraging (see Table 1). The goal of our experiments was to verify the implementability of our approach. We consider traces that go through the loops once and manually generate A and B in SMT-LIB format for input to our tool. These programs contain assertions that can be discharged using loop invariants that are a conjunction of linear inequalities.

First, let us consider the left half of the table. The programs **f1a**, **ex1**, and **f2** are adapted from the benchmarks used in [6]. The programs **nec1** to **nec5** are adapted from NECLA static analysis benchmarks [12]. The program **fse06** is from [7] and is an example on which YOGI [7] does not terminate because it cannot find the invariant $x \geq 0 \wedge y \geq 0$. The program **pldi08**, adapted from [9], requires a disjunction of half-spaces as an invariant. We obtain that by solving the dual problem: we interchange the labels of positive and negative examples and output the negation of the interpolant obtained.

For these examples, we were generating at most ten positive and negative examples before invoking SVM. Hence we expect the column “Total Ex” to have entries less than or equal to 20. Most entries are strictly less than twenty because several predicates have strictly less than ten satisfying assignments. This is expected for A as it represents reachable states and we are considering only one iteration of the loops. So very few states are reachable and hence A has

Table 1. File is the name of the benchmark, LOC is lines of code, Interpolant is the computed interpolant, Total Ex is the sum of the number of positive and negative examples generated for the first iteration of INTERPOLANT. For the second part, Iters represents the number of iterations of INTERPOLANT.

File	LOC	Interpolant	Total Ex	Time (s)	Interpolant	Iter	Time (s)
f1a	20	$x = y$	12	0.017	$x = y \& y \geq 0$	4	0.017
ex1	22	$xa + 2*ya \geq 0$	13	0.019	$xa + 2*ya \geq 0$	4	0.02
f2	18	$3*x \geq y$	13	0.021	$3*x \geq y$	12	0.022
nec1	17	$x \leq 8$	19	0.015	$x \leq 8$	9	0.02
nec2	22	$x < y$	12	0.014	$x < y$	2	0.019
nec3	15	$y \leq 9$	11	0.014	$y \leq 9$	1	0.012
nec4	22	$x = y$	20	0.019	$x = y$	4	0.017
nec5	9	$s \geq 0$	11	0.013	$s \geq 0$	1	0.016
pldi08	10	$x < 0 \mid y > 0$	17	0.02	$6*x < y$	1	0.013
fse06	8	$y \geq 0 \& x \geq 0$	11	0.014	$y \geq 0 \& x \geq 0$	2	0.015

very few satisfying assignments. Nevertheless, 11 to 20 examples are sufficient to terminate INTERPOLANT in a single iteration for all the benchmarks.

To get more intuition about INTERPOLANT, we generate the second part of the table. Here we start with one positive and one negative example. If the classifier is not an interpolant then we add one new point that the classifier misclassifies. The general trend is that we are able to find the same classifier with a smaller number of samples and few iterations. In **f1a** we generate a predicate with more inequalities. This demonstrates that the generated classifier from SVM-I might be sensitive to the order in which misclassified examples are traversed (Fig. 5). For **pldi08**, when we found the classifier between the first positive and negative example generated by Z3 then we found that it was an interpolant. Since the classifier has been generated using only two examples, the training data is insufficient to reflect the full structure of the problem, and unsurprisingly we obtain a predicate that does not generalize well. These experiments suggest that the convergence of INTERPOLANT is faster and the results are better if we start with a reasonable number of samples.

Finally, we compare with the interpolation procedure implemented within OPENSMPT [16] in Table 2. OPENSMPT fails to find the predicate representing the loop invariant for **f1a**, **pldi08**, and **fse06**, whereas our technique succeeds for these examples; this is in line with our claim that machine learning algorithms can provide relevant predicates. OPENSMPT fails on **nec1** because this benchmark contains non-linear multiplications. It turns out that the program has a linear interpolant, found by our technique, which is sufficient to discharge the assertions in the program. Finally, the timing measurements show that we are competitive with OPENSMPT.

Table 2. File is the name of the benchmark and Interpolant is the interpolant computed by the interpolation procedure implemented within OPENSMIT. SAME refers to the benchmarks for which interpolants computed by OPENSMIT were identical to those computed by our technique.

File	Time(s)	Interpolant
f1a	0.022	($y = 1 \mid x \leq 0$) $\&$ $x = 1$) \mid ($y = 0 \& (y = 1 \mid x \leq 0)$)
ex1	0.021	$xa + 2*ya \geq 0 \mid xa + 2*ya \geq 5 \mid xa + 2*ya \geq 5$
f2	0.020	$y \leq 3*x \mid y \leq 3*x + 1 \mid y \leq 3*x + 1$
nec1	NA	FAIL
nec2	0.018	$x < y$ (SAME)
nec3	0.016	$y \leq 9$ (SAME)
nec4	0.021	($x = y \mid y = 0$) \mid ($y = x \mid y = x$)
nec5	0.018	$s \geq 0$ (SAME)
pldi08	0.017	$y > x$
fse06	0.017	$y + x \geq 0 \& y \geq 0 \& y \geq 0 \& y \geq 0$

6 Related Work

In this section, we place our work in the context of existing work on interpolation and machine learning. Our philosophy of computing interpolants from samples is similar to Daikon [5]; Daikon computes likely invariants from program tests. Whereas, we compute sound interpolants statically.

We have considered interpolation only over the quantifier free theory of linear arithmetic. Extension to richer theories, such as the theory of arrays, is left for future work. The interpolants found by our technique are limited to conjunctions of linear inequalities. To handle programs requiring interpolants which are a combination of disjunctions and conjunctions of linear inequalities, we propose to use the existing techniques for control flow refinement [18,26]. These techniques perform source to source semantics preserving transformations so that the loops in the resulting program require only disjunction-free invariants.

Extending the work of [14,22], McMillan [17] computed interpolants of (A, B) , where A and B are in the quantifier free theory of linear arithmetic, in a linear scan of the proof of unsatisfiability of $A \wedge B$. This method requires an explicit construction of the proof of unsatisfiability. In a recent work, Kupferschmid et al. [15] gave a proof based method for finding Craig interpolants for non-linear predicates. The proof based methods like these are generally not scalable: Rybalchenko et al. [24] remark that “Explicit construction of such proofs is a difficult task, which hinders the practical applicability of interpolants for verification.” Like our approach, their method for interpolation is also not proof based. They apply linear programming to find separating hyperplanes between A and B . In contrast to their approach, we are working with samples and not symbolic constraints. This allows us to use mature machine learning techniques like SVMs as well as gives us the ability to handle superficial non-linearities.

We selected SVM for classification as they are one of the simplest and most widely used machine learning algorithms. There are some classification

techniques which are even simpler than SVM [10]. We discuss them here and give the reasons behind not using them for classification. In linear regression, we construct a quadratic penalty term for misclassification and find the hyperplane which minimizes the penalty. Unfortunately the classifiers obtained might err on the training data even if it is linearly separable. Another widespread technique, logistic regression, is guaranteed to find a separating hyperplane if one exists. But the output of logistic regression depends on all examples and hence the output keeps changing even if we add redundant examples. The output of SVMs, on the other hand, is entirely governed by the support vectors and is not affected by other points at all. This results in a robust classifier which is not easily perturbed and leads to better predictability in results.

There has been research on finding non-linear invariants [25,20,23]. These techniques aim at finding invariants which are restricted to polynomials of variables. In contrast, we are not generating non-linear predicates. We are finding linear over-approximations of non-linear constraints and hence our technique only generates linear predicates. On the other hand, unlike [25,20,23] we are not restricted to non-linearities resulting only from polynomials and have demonstrated our technique on an example with transcendental functions.

7 Conclusion

We have shown that classification based machine learning algorithms can be profitably used to compute interpolants and therefore are useful in the context of program verification. In particular, we have given a step-by-step account of how off-the-shelf SVM algorithms can be used to compute interpolants in a sound way. We have also demonstrated the feasibility of applying our approach via experiments over small programs from the literature. Moreover, we are also able to compute interpolants for programs that are not analyzable by existing approaches – specifically, our technique can handle superficial non-linearities.

As future work, we would like to extend our algorithms to compute interpolants for non-linear formulas. We believe that SVMs are a natural tool for this generalization as they have been extensively used to find non-linear classifiers. We would also like to integrate our SVM-based interpolation algorithm with a verification tool and perform a more extensive evaluation of our approach.

Acknowledgements. We thank the anonymous reviewers for their constructive comments. We thank David Dill, Bharath Hariharan, Anshul Mittal, Prateek Jain, and Hristo Paskov for helpful discussions.

References

1. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: EMSOFT, pp. 49–58 (2009)
2. Blum, A., Rivest, R.L.: Training a 3-node Neural Network is NP-Complete. In: Hanson, S.J., Rivest, R.L., Remmle, W. (eds.) MIT-Siemens 1993. LNCS, vol. 661, pp. 9–28. Springer, Heidelberg (1993)

3. Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2, 27:1–27:27 (2011), software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>
4. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.* 22(3), 269–285 (1957)
5. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69(1-3), 35–45 (2007)
6. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically Refining Abstract Interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
7. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE, pp. 117–127 (2006)
8. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI, pp. 375–385 (2009)
9. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, pp. 281–292 (2008)
10. Hastie, T., Tibshirani, R., Friedman, J.: *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc. (2001)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244 (2004)
12. Ivancic, F., Sankaranarayanan, S.: NECLA Static Analysis Benchmarks, http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz
13. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
14. Krajicek, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symb. Log.* 62(2), 457–486 (1997)
15. Kupferschmid, S., Becker, B.: Craig Interpolation in the Presence of Non-linear Constraints. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 240–255. Springer, Heidelberg (2011)
16. Leroux, J., Rümmer, P.: Craig Interpolation for Presburger Arithmetic in OpenSMT, <http://www.philipp.ruemmer.org/interpolating-opensmt.shtml>
17. McMillan, K.L.: An interpolating theorem prover. *Theor. Comput. Sci.* 345(1), 101–121 (2005)
18. Megiddo, N.: On the complexity of polyhedral separability. *Discrete & Computational Geometry* 3, 325–337 (1988)
19. 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)
20. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. *Inf. Process. Lett.* 91(5), 233–244 (2004)
21. Platt, J.C.: Fast training of support vector machines using sequential minimal optimization. In: *Advances in Kernel Methods: Support Vector Learning*, pp. 185–208. MIT Press (1998)
22. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.* 62(3), 981–998 (1997)

23. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. *J. Symb. Comput.* 42(4), 443–476 (2007)
24. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
25. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL, pp. 318–329 (2004)
26. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying Loop Invariant Generation Using Splitter Predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011)

Termination Analysis with Algorithmic Learning[☆]

Wonchan Lee¹, Bow-Yaw Wang², and Kwangkeun Yi¹

¹ Seoul National University, Korea

² Academia Sinica, Taiwan

Abstract. An algorithmic-learning-based termination analysis technique is presented. The new technique combines transition predicate abstraction, algorithmic learning, and decision procedures to compute transition invariants as proofs of program termination. Compared to the previous approaches that mostly aim to find a particular form of transition invariants, our technique does not commit to any particular one. For the examples that the previous approaches simply give up and report failure our technique can still prove the termination. We compare our technique with others on several benchmarks from literature including POLYRANK examples, SNU realtime benchmark, and Windows device driver examples. The result shows that our technique outperforms others both in efficiency and effectiveness.

1 Introduction

Termination is a critical property of functions in program libraries. Invoking a non-terminating library function may result in system lagging or even freezing. Because of its importance, termination analysis has been studied extensively [2–4, 8, 10–14, 17, 22, 24–27] for the last decade and advanced to the level of industrial uses [1, 13].

Among various strategies for proving termination, we are most interested in the transition invariant-based technique. A transition invariant for a transition relation is an over-approximation to the reachable transitive closure of the transition relation [13, 25, 26]. Podelski and Rybalchenko [25] have shown that the termination of a program amounts to the existence of a disjunctively well-founded transition invariant for its transition relation. We therefore aim to find a disjunctively well-founded transition invariant for the transition relation of a program.

Though transition invariants can be defined as a fixpoint, they are not necessarily computed by costly fixpoint iterations. Observe that it suffices to find one disjunctively well-founded *over-approximation* to the least reachable transition

* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2012-0000468) and National Science Council of Taiwan Grant Numbers 99-2218-E-001-002-MY3 and 100-2221-E-002-116-.

invariant. If there are lots of such over-approximations, we have only to design an efficient algorithm to compute one. Indeed, several such algorithms have been proposed to compute reachable transition invariants efficiently [2, 8, 22, 27].

In this paper, we report the first algorithmic-learning-based technique for termination analysis. Recently, algorithmic learning is successfully applied to avoid the costly fixpoint iteration in the context of loop invariant generation [19–21]. In the same spirit, the technique we propose in this paper finds disjunctively well-founded transition invariants without the excessive cost of fixpoint iterations by combining algorithmic learning, transition predicate abstraction, decision procedures, and well-foundedness checkers. Through transition predicate abstraction, we adopt a learning algorithm for Boolean formulae to infer transition invariants over given atomic predicates. Using an SMT solver and well-foundedness checker, we design a mechanical teacher to guide the learning algorithm to find a disjunctively well-founded transition invariant. Randomness is moreover employed to exploit the multitude of transition invariants.

The advantage of our technique is that it can be both efficient and effective, compared to the previous works [2, 8, 22, 27]. The key innovation of our technique is that we decouple the construction of transition invariants from the transition predicate generation. In previous works, the transition predicate generation is tightly coupled with the transition invariant inference and the whole process is optimized by committing to a particular form of transition invariants, which might hurt the effectiveness. However, the following intuition from the years of research on termination analysis teaches us that this is not necessarily the case; termination arguments, or transition predicates, are evident in most cases [27], but it is not so obvious how to combine those predicates to get a disjunctively well-founded transition invariant [2]. To solve the “not-so-obvious” problem efficiently, we use algorithmic learning which was proven to work well in a different domain, inferring loop invariants out of atomic predicates. Being trained by mechanical teachers, learning algorithms become an efficient engine for exploring possible combinations of predicates. For the atomic transition predicate generation, we employ a simple heuristic, which is turned out to be effective for most of examples in the experiments. We can further improve the effectiveness with additional predicates.

Example. Consider the following nested loop. We found that this simple nested loop cannot be proven by any of the existing termination analysis tools [8, 22, 27]:

```
while i < 10 do {j ← 0; while j < 10 do {i', j' ← i + 1, j + 1}}
```

Our simple heuristic finds the set $\{i < 10, j < 10, i < i', j < j'\}$ of atomic transition predicates. Then, our randomized technique first computes a transition invariant for the inner loop, say, $j < 10 \wedge j < j'$. Since the transition invariant is well-founded, it proves the termination of the inner loop. Next, we replace the inner loop by its transition invariant and proceed to find a transition invariant for the following simple loop:

```
while i < 10 do {j ← 0; assume(j < 10 \wedge j < j');}
```

Since its loop body does not update the variable i , it is impossible to prove the termination of the loop. This is exactly what happens in some of the existing tools [8, 27]; after they compute $j < 10 \wedge j < j'$ as a transition invariant of the inner loop, they simply report possible non-termination of the outer loop. The other tool [22] fails because it uses a even more imprecise transition invariant, *true*, as the summary of the inner loop; the tool unrolls and unions the transition relation of a loop body until it reaches a transition invariant and when it unrolls the outer loop, the tool can only assume that the inner loop can change variables arbitrarily. However, there exists another transition invariant $j < 10 \wedge j < j' \wedge i < i'$ which are both expressible with the given predicates and precise enough to prove the termination of the outer loop. As long as a transition invariant is expressible with the given predicates, our randomized algorithm for termination analysis can find it. Let us say our technique returns $j < 10 \wedge j < j' \wedge i < i'$ this time. The new transition invariant is again well-founded. We proceed to replace the inner loop by the new transition invariant:

```
while  $i < 10$  do { $j \leftarrow 0$ ; assume( $j < 10 \wedge j < j' \wedge i < i'$ ); }
```

Our termination analysis algorithm is now able to infer the transition invariant $i < 10 \wedge i < i'$ for the simple loop. Since the transition invariant $i < 10 \wedge i < i'$ is well-founded, we conclude that the outer loop is terminating as well. \square

Contributions.

- We design and implement an algorithmic-learning-based termination analyzer. As far as we know, our work is the first to apply the algorithmic learning to termination analysis problem.
- We empirically show that the prototype implementation of our technique outperforms the previous tools [8, 22, 27] both in efficiency and effectiveness.

Organization. Section 2 reviews termination analysis via transition invariants and presents our formalism of transition invariants in intentional representation. Section 3 explains algorithmic-learning-based inference approach and how to apply it to the problem of inferring disjunctively well-founded transition invariants. Section 4 presents our experiment results. Section 5 discusses related work and Section 6 concludes.

2 Termination Analysis via Transition Invariants

This section explains the termination analysis technique based on transition invariants. The technique was first introduced by Podelski and Rybalchenko [25] and later implemented on top of the SLAM model checker [13]. We first review the original theory of transition invariants in an extensional view [18]. We then present our formalism of transition invariants in an intensional view which we compute via algorithmic-learning-based approach.

2.1 Program Termination and Transition Invariant

A program $P = \langle W_P, I_P, R_P \rangle$ consists of a set W_P of states, a set $I_P \subseteq W_P$ of initial states, and a transition relation $R_P \subseteq W_P \times W_P$.

A program P terminates if there is no infinite sequence s_1, s_2, \dots of states such that $s_1 \in I_P$ and $(s_i, s_{i+1}) \in R_P$. This condition is equivalent to the well-foundedness of $R_P \cap \text{Reach}(P) \times \text{Reach}(P)$. Here, the set $\text{Reach}(P)$ denotes the set of reachable states.

Instead of showing $R_P \cap \text{Reach}(P) \times \text{Reach}(P)$ is well-founded, we prove the termination by finding its disjunctively well-founded transition invariant [25]. A transition invariant T of P is a relation that contains a reachable portion of the transitive closure of R_P :

$$R_P^+ \cap \text{Reach}(P) \times \text{Reach}(P) \subseteq T.$$

Furthermore, we say the transition invariant T is disjunctively well-founded when it is a union of a finite number of well-founded relations T_1, \dots, T_N .

Theorem 1 ([25]). *A program P terminates if and only if there exists a disjunctively well-founded transition invariant T of P .*

Thanks to Theorem 1, the problem of program termination now becomes finding a disjunctively well-founded transition invariant for a given program P .

Cook et al. [13] showed that transition invariants can be reduced to reachability analysis. The authors named the relation $R_P^+ \cap \text{Reach}(P) \times \text{Reach}(P)$ *binary reachability relation*, which is the least fixpoint of the following functional F_P starting from the relation \perp_P .

$$\begin{aligned} F_P(X) &\triangleq (X \cup id_{(2)}(X)) \circ R_P \\ id_{(2)}(X) &\triangleq \{(\nu_2, \nu_2) \in W_P \times W_P : \exists \nu_1. (\nu_1, \nu_2) \in X\} \\ X \circ Y &\triangleq \{(\nu_1, \nu_3) \in W_P \times W_P : \exists \nu_2. (\nu_1, \nu_2) \in X \text{ and } (\nu_2, \nu_3) \in Y\} \\ \perp_P &\triangleq \{(\nu_1, \nu_2) \in W_P \times W_P : \nu_1 \in I_P \text{ and } (\nu_1, \nu_2) \in R_P\} \end{aligned}$$

In the next subsection, we show how to compute an over-approximation of this binary reachability relation via intensional representations of transition invariants.

2.2 Intensional Transition Invariants

Simple Loop Programs. For presentation, we consider a simple loop program P with the following abstract syntax.

$$\begin{aligned} P &::= \{l\} \text{ while } l \text{ do } S \\ S &::= v \leftarrow e \mid v \leftarrow \text{nondet} \mid \text{assume } l \mid S \sqcup S \mid S; S \\ l &::= e \leq n \mid l \wedge l \mid l \vee l \\ e &::= n \mid v \mid n \times e \mid e + e \mid e - e \quad (v \in V, n \in \mathbb{Z}) \end{aligned}$$

where V and \mathbb{Z} is a set of variables and integers respectively, and l represents quantifier-free formulae over integer affine predicates. A loop with a loop guard

is annotated with a formula specifying a precondition. We write κ_P for the loop guard and δ_P for the precondition. In the syntax, we have non-deterministic assignments ($v \leftarrow \text{nondet}$) to emulate the behaviors of unsupported features such as arrays or function calls. For brevity, we use choice ($S \sqcup S$) and assume statements ($\text{assume } l$) instead of traditional if statements.

A program state $\nu \in W_P$ of the program P is a map from V to \mathbb{Z} . Given a formula l , we write $\nu \models_{\text{sat}} l$ when ν satisfies l . We write $\models_{\text{sat}} l$ if there exists a state that satisfies l . When the formula is satisfied by all states, we write $\models l$. We define the set $\mathcal{W}(l)$ to be $\{\nu \in W_P : \nu \models_{\text{sat}} l\}$. For a simple loop program P , the set I_P of initial states is the same as $\mathcal{W}(\delta_P)$.

To describe the transition relation R_P for a simple loop program, we define transition semantics $\llbracket P \rrbracket$ of P . The transition semantics is a quantifier-free formula over sets V and V' describing the current state and the state after the transition, respectively. The transition semantics is defined as follows:

$$\begin{aligned} \llbracket \{\delta_P\} \text{ while } \kappa_P \text{ do } S \rrbracket &\triangleq \kappa_P \wedge \llbracket S \rrbracket \\ \llbracket v \leftarrow e \rrbracket &\triangleq v' = e \wedge \bigwedge_{w \in V \setminus \{v\}} w' = w \\ \llbracket v \leftarrow \text{nondet} \rrbracket &\triangleq v' = v'' \wedge \bigwedge_{w \in V \setminus \{v\}} w' = w \quad (v'' : \text{fresh}) \\ \llbracket \text{assume } l \rrbracket &\triangleq l \wedge \bigwedge_{w \in V \setminus Vars(l)} w' = w \\ \llbracket S_0 \sqcup S_1 \rrbracket &\triangleq \llbracket S_0 \rrbracket \vee \llbracket S_1 \rrbracket \\ \llbracket S_0; S_1 \rrbracket &\triangleq \llbracket S_0 \rrbracket[V' \mapsto V''] \wedge \llbracket S_1 \rrbracket[V \mapsto V''] \quad (V'' : \text{fresh}) \end{aligned}$$

where $Vars(l)$ is the set of variables appeared in l and $f[v_1 \mapsto v_2]$ is the formula obtained by substituting the variable v_2 for v_1 in f . Given a formula f over V and V' , we write $\nu, \nu' \models_{\text{sat}} f$ when the formula obtained by replacing $v \in V$ in f with $\nu(v)$ and $v' \in V'$ in f with $\nu'(v')$ is satisfiable. The notations $\models_{\text{sat}} f$ and $\models f$ are defined accordingly. The notation $\mathcal{R}(f)$ denotes the relation $\{(\nu, \nu') \in W_P \times W_P : \nu, \nu' \models_{\text{sat}} f\}$. Thus the transition relation R_P of a simple loop program P is $\mathcal{R}(\llbracket P \rrbracket)$.

In summary, a simple loop program P defines the program $\langle W_P, \mathcal{W}(\delta_P), \mathcal{R}(\llbracket P \rrbracket) \rangle$.

Intensional Transition Invariants. For a simple loop program P , we define the intensional representations of the functional F_P and the relation \perp_P (written F_P^\sharp and \perp_P^\sharp respectively) as follows:

$$\begin{aligned} F_P^\sharp(f) &\triangleq (f \vee id_{(2)}^\sharp(f)) \circ^\sharp \llbracket P \rrbracket \\ id_{(2)}^\sharp(f) &\triangleq f[V \mapsto V''] \wedge V = V' \quad (V'' : \text{fresh}) \\ f \circ^\sharp g &\triangleq f[V' \mapsto V''] \wedge g[V \mapsto V''] \quad (V'' : \text{fresh}) \\ \perp_P^\sharp &\triangleq \delta_P \wedge \llbracket P \rrbracket \end{aligned}$$

where $f[\{v_1, \dots, v_n\} \mapsto \{v'_1, \dots, v'_n\}] \triangleq f[v_1 \mapsto v'_1] \cdots [v_n \mapsto v'_n]$. The following lemmas show that F_P^\sharp and \perp_P^\sharp correspond to F_P and \perp_P respectively.

Lemma 1. *For any simple loop program P , $\mathcal{W}(\perp_P^\sharp) = \perp_P$.*

Lemma 2. *Let f be a quantifier-free formula over V and V' . For any simple loop program P , $\mathcal{R}(F_P^\sharp(f)) = F_P(\mathcal{R}(f))$.*

From the properties of F_P^\sharp and \perp_P^\sharp , we compute a transition invariant of a simple loop program P by finding a formula \mathcal{T} that satisfies the following conditions:

1. $\models \perp_P^\sharp \implies \mathcal{T}$;
2. $\models \mathcal{T} \implies \kappa_P$;
3. $\models F_P^\sharp(\mathcal{T}) \implies \mathcal{T}$.

The first condition is to guarantee that \mathcal{T} subsumes the first iteration starting from the initial state. The second condition is to guarantee that \mathcal{T} expresses only the iterations within the loop. The last condition is to guarantee that \mathcal{T} is a fixpoint. Note that this fixpoint is not necessarily a *least* fixpoint. We call the formula \mathcal{T} *intensional* transition invariant which is an intensional representation of a transition invariant.

Lemma 3. *Let \mathcal{T} be an intensional transition invariant of a simple loop program P . Then $\mathcal{R}(\mathcal{T})$ is a transition invariant; i.e. $\mathcal{R}(\mathcal{T}) \supseteq R_P^+ \cap \text{Reach}(P) \times \text{Reach}(P)$.*

We say \mathcal{T} is disjunctively well-founded when $\mathcal{R}(\mathcal{T})$ is disjunctively well-founded. Disjunctively well-founded intensional transition invariants are proofs of program termination.

Theorem 2. *A simple loop program P terminates if there exists a disjunctively well-founded intensional transition invariant \mathcal{T} of P .*

In the rest of the paper, transition invariants mean intensional transition invariants unless stated otherwise.

3 Algorithmic-Learning-Based Inference of Transition Invariants

The key idea of the algorithmic-learning-based framework [19–21] is to apply CDNF algorithm [4] to infer a formula with a mechanical teacher. CDNF algorithm is an exact learning algorithm for Boolean formulae. It infers an arbitrary Boolean formula over fixed variables by interacting with a teacher. In our case, we are particularly interested in finding transition invariants over the given set of atomic transition predicates. In order to apply CDNF algorithm, we will design a mechanical teacher to guide the learning algorithm to infer a transition invariant for a simple loop program.

In this section, we explain our design of the mechanical teacher in details. We first introduce CDNF algorithm for Boolean formulae. Through transition predicate abstraction, the correspondence between Boolean formulae and quantifier-free formulae over atomic transition predicates is explained. Lastly, we present our design of the mechanical teacher.

3.1 CDNF Learning Algorithm

CDNF algorithm is an exact learning algorithm for Boolean formulae. It infers an unknown target formula by posing queries to a teacher. The teacher is responsible for answering two types of queries. The learning algorithm may ask if a valuation satisfies the target formula by a membership query. Or it may ask if a conjectured formula is equivalent to the target in an equivalence query. According to the answers to queries, CDNF algorithm will infer a Boolean formula equivalent to the unknown target within a polynomial number of queries in the formula size of the target.

In order to apply CDNF algorithm, a mechanical teacher that answers queries from the learning algorithm is needed. The mechanical teacher consists of two algorithms. The membership query resolution algorithm (*MEM*) answers membership queries; the equivalence query resolution algorithm (*EQ*) resolves equivalence queries. The algorithm *MEM* returns *YES* if the given valuation satisfies the unknown target and *NO* otherwise. The algorithm *EQ* returns *YES* if the given conjecture is equivalent to the target and a counterexample otherwise. Let \mathbf{x} be a set of Boolean variables, and $BF[\mathbf{x}]$ and $Val_{\mathbf{x}}$ be the set of Boolean formulae and valuations over \mathbf{x} , respectively. The signatures of these query resolution algorithms are as follows:

$$\begin{aligned} MEM : Val_{\mathbf{x}} &\rightarrow \{ YES, NO \} \\ EQ : BF[\mathbf{x}] &\rightarrow \{ YES \} + Val_{\mathbf{x}} \end{aligned}$$

3.2 Learning Algorithm as an Inference Engine

We establish a connection between Boolean formulae and quantifier-free formulae. The connection enables CDNF algorithm to infer transition invariants. We consider transition predicate abstraction [26] over a set \mathcal{P} of atomic predicates defined over V and V' . A quantifier-free formula f over \mathcal{P} is generated by the following syntax.

$$f ::= p \mid \neg f \mid f \vee f \mid f \wedge f$$

where $p \in \mathcal{P}$. We write $QF[\mathcal{P}]$ for the set of quantifier-free formulae over \mathcal{P} . The set $QF[\mathcal{P}]$ and the set $BF[\mathbf{x}]$ of Boolean formulae, where $\mathbf{x} = \{x_{p_i} \mid p_i \in \mathcal{P}\}$, establishes the following Galois connection.

$$QF[\mathcal{P}] \xrightleftharpoons[\alpha]{\gamma} BF[\mathbf{x}]$$

From the connection, we know that once the learning algorithm finds a Boolean formula, then it has a corresponding quantifier-free formula that we want to find.

We now show how to make a mechanical teacher under the transition predicate abstraction. We define the following two functions $\bar{\alpha}$ and $\bar{\gamma}$ that translate valuations over V and \mathbf{x} , respectively.

$$\begin{aligned} \bar{\alpha}(\nu, \nu') &\triangleq \mu \text{ such that } \mu \models \bigwedge_{\nu, \nu' \models_{\text{sat}} p} x_p \wedge \bigwedge_{\nu, \nu' \not\models_{\text{sat}} p} \neg x_p \\ \bar{\gamma}(\mu) &\triangleq \bigwedge_{\mu(x_p)=\top} p \wedge \bigwedge_{\mu(x_p)=\perp} \neg p. \end{aligned}$$

The design of the query resolution algorithms MEM and EQ amounts to that of two concrete algorithms MEM^\sharp and EQ^\sharp with the following signatures:

$$\begin{aligned} MEM^\sharp : QF[\mathcal{P}] &\rightarrow \{ YES, NO \} \\ EQ^\sharp : QF[\mathcal{P}] &\rightarrow \{ YES \} + Val_V \times Val_V \end{aligned}$$

With the two concrete algorithms and the translation functions $\overline{\alpha}$ and $\overline{\gamma}$, we derive the query resolution algorithms MEM and EQ as follows:

$$\begin{aligned} MEM(\mu) &= MEM^\sharp(\overline{\gamma}(\mu)) \\ EQ(b) &= \begin{cases} \overline{\alpha}(\nu, \nu') & \text{when } EQ^\sharp(\gamma(b)) = (\nu, \nu') \\ YES & \text{otherwise} \end{cases} \end{aligned}$$

3.3 Algorithms for Mechanical Teacher

In the rest of this section, we explain how to design the algorithms MEM^\sharp and EQ^\sharp for transition invariants. One technical question is how we can make MEM^\sharp and EQ^\sharp answer questions on the formula that we do not know yet. We solve this problem simply by giving random answers when we cannot answer conclusively. Interesting observation is that as far as those answers are consistent and algorithm EQ^\sharp returns YES when it really finds the one, CDNF algorithm can still infer the target formula. We exploit the fact that there can exist multiple formulae that are equivalent to the target.

Membership Query Resolution. In a membership query $MEM(\mu)$ with $\mu \in Val_x$, we would like to know if μ is included in an unknown target Boolean formula that represents a disjunctively well-founded transition invariant. Since we do not know any disjunctively well-founded transition invariant yet, we can not answer every membership query conclusively.

To see what amount of answers we can give conclusively, we first consider the conditions that μ should respect. Suppose \mathcal{T} is a transition invariant. If μ satisfies the target Boolean formula, we have $\overline{\gamma}(\mu) \implies \mathcal{T}$. Moreover, we have $\perp_P^\sharp \implies \mathcal{T} \implies \kappa_P$ for \mathcal{T} is a transition invariant. Therefore, we have the following relationship:

1. If $\not\models \overline{\gamma}(\mu) \implies \kappa_P$, then $\not\models \overline{\gamma}(\mu) \implies \mathcal{T}$;
2. If $\models \overline{\gamma}(\mu) \implies \perp_P^\sharp$, then $\models \overline{\gamma}(\mu) \implies \mathcal{T}$.

In the first case, we can conclusively answer NO . Similarly, we answer YES for the second case conclusively.

For the others cases, we can give random answers. Since we are looking for disjunctively well-founded transition invariants, we heuristically answer NO when $\mathcal{R}(\overline{\gamma}(\mu))$ is not well-founded.

Algorithm \blacksquare summarizes the membership query resolution. The $MEM^\sharp(f)$ algorithm first checks if $\models f \implies \kappa_P$. If not, it returns NO . The algorithm then checks if $\mathcal{R}(f)$ is well-founded. If not, it heuristically returns NO . Finally, the algorithm checks if $\models f \implies \perp_P^\sharp$. If so, it returns YES since we know for sure

Algorithm 1. $MEM^\sharp(f)$

```

Input:  $f \in QF[\mathcal{P}]$ 
Output: YES or NO
1 if  $\models_{sat} f \wedge \neg \kappa_P$  then
2   | return NO
3 else
4   | if  $\mathcal{R}(f)$  is well-founded then
5     |   | if  $\models_{sat} f \wedge \neg \perp_P^\sharp$  then
6       |     |   | return YES or NO randomly
7     |   | else
8       |     |   | return YES
9   | else
10  |   | return NO

```

Algorithm 2. $EQ^\sharp(f)$

```

Input:  $f$  : a CDNF formula such that  $f = \bigwedge_{i=1}^n f_i$ 
Output: YES or a counterexample  $(\nu, \nu') \in Val_V \times Val_V$ 
1 if  $isTransitionInvariant(f)$  is  $(\nu, \nu') \in Val_V \times Val_V$  then
2   | return  $(\nu, \nu')$ 
  //  $f$  is a transition invariant
3 if  $hasDWFCConjunct(f)$  is YES then
4   | return YES
  //  $f_i$  is not disjunctively well-founded for every  $i$ 
5 if  $findCounterexample(f)$  is  $(\nu, \nu') \in Val_V \times Val_V$  then
6   | return  $(\nu, \nu')$ 
7 restart CDNF algorithm

```

that $\overline{\gamma}^{-1}(f)$ is the member of the target formula. Otherwise, it gives a random answer to the learning algorithm.

Equivalence Query Resolution. In an equivalence query $EQ(b)$, we are given a CDNF formula b over \mathbf{x} as the conjecture. The algorithm should check whether $\gamma(b)$ is a disjunctively well-founded transition invariant for the simple loop program P . If not, it returns a valuation over \mathbf{x} as a counterexample.

Algorithm 2 presents the equivalence query resolution algorithm $EQ^\sharp(f)$. The algorithm first checks if f is a transition invariant. If not, it returns a counterexample. Next, it checks if the formula has a disjunctively well-founded conjunct f_i . If so, we have found a disjunctively well-founded transition invariant. Otherwise, the algorithm tries to find a counterexample that possibly makes the formula not disjunctively well-founded. If it cannot find a counterexample, the algorithm simply restarts to find another transition invariant.

Invariance Check. Algorithm 3 shows the procedure to check if f satisfies the three conditions of transition invariants. If the conjecture f does not satisfy one of them, the algorithm returns a counterexample.

Algorithm 3. *isTransitionInvariant(f)*

Input: f : a CDNF formula
Output: YES or a counterexample $(\nu, \nu') \in Val_V \times Val_V$

```

1 if  $\models \perp_P^\sharp \implies f$  and  $\models f \implies \kappa_P$  and  $\models F_P^\sharp(f) \implies f$  then
2   | return YES
3 if  $\nu, \nu' \models_{sat} \perp_P^\sharp \wedge \neg f$  then
4   | return  $(\nu, \nu')$ 
5 if  $\nu, \nu' \models_{sat} f \wedge \neg \kappa_P$  then
6   | return  $(\nu, \nu')$ 
7 if  $\nu, \nu' \models_{sat} F_P^\sharp(f) \wedge \neg f$  then
8   | return  $(\nu, \nu')$ 

```

Algorithm 4. *hasDWFConjunct(f)*

Input: f : a CDNF formula such that $f = \bigwedge_{i=1}^n \bigvee_{j=1}^{m_i} f_{ij}$
Output: YES if f_i is disjunctively well-founded for some i ; NO otherwise

```

1 foreach  $i = 1, \dots, n$  do
2   | isWellFounded  $\leftarrow \top$ 
3   | foreach  $j = 1, \dots, m_i$  do
4     |   | if  $\mathcal{R}(f_{ij})$  is not well-founded then
5       |       | isWellFounded  $\leftarrow \perp$ 
6       |       | break
7   |   if isWellFounded then return YES
8 return NO;

```

Disjunctively Well-foundedness Check. Algorithm 4 checks if f_i is disjunctively well-founded for some i . Recall that f is a CDNF formula such that $f = \bigwedge_{i=1}^n f_i$ and each f_i is also a transition invariant since f implies f_i . If the algorithm has found one disjunctively well-founded f_i , we have found a disjunctively well-founded transition invariant. The following lemma states the correctness of the algorithm.

Lemma 4. *Let $f = \bigwedge_{i=1}^n f_i$ be a CDNF formula. If f is a transition invariant and f_i is disjunctively well-founded for some i , f_i is a disjunctively well-founded transition invariant.*

For each DNF formula f_i , we check if all of its disjuncts are well-founded. Each disjunct f_{ij} is a conjunction of atomic transition predicates and we can check the well-foundedness using existing well-foundness checkers [4, 5, 11, 24].

Counterexample Generation. Conjectures from learning algorithms are sometimes not disjunctively well-founded even if they are a transition invariant. Those are either containing an idle transition, which does nothing during an iteration, or the ones that become disjunctively well-founded once proper bound conditions are added. For example, transition invariant $x' \leq x$ contains an idle transition and transition invariant $x' < x$ becomes well-founded if additional bound condition $x > 0$ is added. We implemented an algorithm that generates a

Algorithm 5. *findCounterexample(f)*

Input: f : a CDNF formula
Output: a counterexample $(\nu, \nu') \in Val_V \times Val_V$ or *FAIL*

- 1 **if** $(\nu, \nu') \models_{sat} f \wedge V' = V$ **then return** (ν, ν') *as a counterexample*
- 2 **return** *FAIL*

Algorithm 6. Pseudo-code of the main loop

Input: set \mathcal{P} of transition predicates

- 1 **while** there exists a simple loop P in the program **do**
- 2 repeat N times to infer d.wf transition invariant $\mathcal{T} \in QF[\mathcal{P}]$ using CDNF algorithm
- 3 **if** \mathcal{T} is found **then**
- 4 | replace P with $\text{assume}(\kappa_P \wedge \mathcal{T} \vee \neg \kappa_P \wedge V = V');$
- 5 **else**
- 6 | replace P with $\text{assume}(\kappa_P \wedge \text{true} \vee \neg \kappa_P \wedge V = V');$

counterexample for both cases, but for space reason, we present in Algorithm 5 a simplified procedure that handles only the first case. If the algorithm finds an idle transition ($\models_{sat} f \wedge V' = V$), it returns a counterexample. Otherwise it returns *FAIL*, hoping that the learning algorithm finds another formula next time.

4 Experiments

To evaluate our approach, we implemented our algorithm and compared it with existing tools. In the implementation, we use Z3 SMT solver [16] for satisfiability check and our own implementation of RANKFINDER algorithm [24] for well-foundedness check.

Algorithm 6 shows the pseudo-code of the main loop of our analyzer. The algorithm essentially handles the nested loop in the manner similar to that of [27]; it finds a non-nested simple loop and tries to find a disjunctively well-founded transition invariant; when it finds one, we can use it as a summary of the loop and make the outer-loop also non-nested; if the inference fails within the given limit N , it simply assumes that the loop can change the variable arbitrarily and uses *true* as its summary.

In Algorithm 6, we make CDNF algorithm repeat only a certain number of times because the learning algorithm loops indefinitely if a given loop does not terminate or it does but there is no disjunctively well-founded transition invariant expressible with the given set of predicates. In practice, CDNF algorithm could find a disjunctively well-founded transition invariant within several trials.

We implement a simple heuristic that generates atomic transition predicates using loop guards and branch conditions. First, all loop guards and branch conditions are used as atomic transition predicates. Second, for each loop guard, say $E_1 \geq E_2$, we generate predicates $E'_1 - E'_2 < E_1 - E_2$. The intuition is that the gap

between values of E_1 and E_2 should decrease so that the loop guard would be eventually violated. According to our experience, even with this simple heuristic we could verify almost all terminating examples (only four predicates are required to add manually in the whole experiments).

For comparison, we use the following four tools.

- LTA)** Our prototype algorithmic-Learning-based Termination Analyzer (LTA) with a simple heuristic for transition predicate generation.
- LF)** LOOPFROG [27], a summary-based termination analyzer. LOOPFROG can be configured with five different templates of transition invariants and we use only the template $i' \diamond i$ where $\diamond = \{<, >\}$, which showed the best performance according to [27].
- LR)** LINEARRANKTERM [8], an abstract interpretation-based termination analyzer.
- CTA)** Compositional Termination Analyzer (CTA) [22].

All experiments are done on Intel Core i7 3.07 Ghz CPU with 24GB memory running Linux 2.6.35. The timeout for CTA is set to one hour and LTA is configured with the retrial limit (N in Algorithm 6) 100.

In all experiments, we report only the elapsed time for cases that tools could prove the termination. If there are multiple loops, we report the elapsed time aggregated only on terminating cases (denoted by '+' after numbers). The reason is that our technique is semi-algorithm; it is not meaningful to report the elapsed time to eventually fail since it simply depends on the parameter N . We run each case 100 times and take the average of them.

We use four sets of examples¹ from the literature, which are examples from Octagon library [23], POLYRANK distribution [5, 6], Windows device drivers [2, 8], and SNU real-time benchmark suite [27]. Since our prototype supports a fragment of full ANSI-C, some examples are manually translated when they use unsupported features. The experiment results are given in Figure 11.

Figure 11(a) and (b) shows the results on examples from Octagon Library and POLYRANK distribution², respectively. All examples are known to terminate. Our tool is the only one that proves all examples from Octagon library (note that we got a different result from the one in [8]; we tried our best but we could not make LINEARRANKTERM prove example 3). In terms of efficiency, LTA outperforms the others except LOOPFROG; since LOOPFROG considers only one iteration of loops with the pre-defined transition invariant template, it is very efficient for simple programs. For the examples from POLYRANK distribution, only LTA and LINEARRANKTERM can prove the first two.

Figure 11(c) shows the result on examples from Windows device drivers. Examples 2, 3, and 9 are known to have termination bugs and the others terminate. Only LTA and LINEARRANKTERM can prove all the terminating cases and LTA

¹ We made them available at <http://ropas.snu.ac.kr/cav12/>. Windows device driver examples cannot be made available due to the license issue.

² As already noted in [2, 8], there was no example 5 in the original distribution. We used the same numbering to avoid confusion.

	1	2	3	4	5	6
LTA	0.01	0.01	0.59	0.12	0.01	0.03
LF	0.01	0.01	0.02	0.03	∅	0.01
LR	0.20	0.16	∅	0.32	0.21	0.79
CTA	0.58	0.26	9.48	∅	∅	0.48

(a) Results on examples from the Octagon Library

	1	2	3	4	6	7	8	9	10	11	12
LTA	0.03	0.45	∅	∅	∅	∅	∅	∅	∅	∅	∅
LF	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
LR	0.71	0.34	∅	∅	∅	∅	∅	∅	∅	∅	∅
CTA	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅

(b) Results on examples from the POLYRANK distribution

	1	2	3	4	5	6	7	8	9	10
LTA	0.43	∅	∅	0.02	0.01	0.60	0.30	0.20	∅	0.03
LF	∅	∅	∅	0.00	0.00	∅	∅	∅	∅	∅
LR	0.66	∅	∅	0.32	0.16	0.34	0.54	0.29	∅	0.28
CTA	T/O	∅	∅	0.41	0.44	2.04	8.86	8.87	∅	T/O

(c) Results on small arithmetic examples taken from Windows device drivers

Example	Tool	✓	∅	Time
bs	LTA	1	0	0.01
1 loop	LF	0	1	N/A
1 terminates	CTA	0	1	N/A
fft1k	LTA	3	0	0.04
3 loops	LF	2	1	0.03+
3 terminate	CTA	2	1	0.54+
fft1	LTA	2	3	0.03+
5 loops	LF	2	3	0.18+
2 terminate	CTA	2	3	0.66+
insertsort	LTA	1	1	0.01+
2 loops	LF	1	1	0.01+
1 terminates	CTA	1	1	0.29+

Example	Tool	✓	∅	Time
ludcmp	LTA	11	0	0.13
11 loops	LF	5	6	0.07+
11 terminate	CTA	4	7	1.50+
minver	LTA	15	2	0.23+
17 loops	LF	16	1	0.22+
15 terminate	CTA	15	2	5.21+
qsort-exam	LTA	2	4	0.66+
6 loops	LF	0	6	N/A
2 terminate	CTA	0	6	N/A
select	LTA	0	4	N/A
4 loops	LF	0	4	N/A
0 terminates	CTA	0	4	N/A

(d) Results on modified examples from SNU real-time benchmark

Fig. 1. Experiment Results. **LTA** is used to represent algorithmic-learning-based termination analyzer. **LF** is used to represent LOOPFROG, summary-based termination analyzer. **LR** is used to represent LINEARRANKTERM, abstract interpretation-based termination analyzer. **CTA** is used to denote compositional termination analyzer. Symbol '+' means that the time is aggregated only when the tool proved the termination. ✓="termination proven". ∅="termination not proven". N/A="not comparable". T/O="time out". **Tool** and **Time** show the name and the runtime of tools.

shows better performance than LINEARRANKTERM for all examples except example 6.

Figure II(d) is the result on SNU real-time benchmark suite³. The original examples in the suite contain many trivial, non-nested loops of form `for(i=0; i<n; ++i){...}` (52 out of 107 loops). We leave them out and make suite contain only non-trivial, nested loops. We show in the figure the number of terminating loops in each example, which was found manually.

Figure II(d) shows that LTA outperforms LOOPFROG and CTA, both in efficiency and effectiveness. Note that there is no comparison between LINEARRANKTERM and LTA; we could not compare LTA with LINEARRANKTERM on the examples that have non-terminating loops since LINEARRANKTERM stops the analysis as soon as it finds any single termination bug. We report here the results on the examples with terminating loops only; for three such examples (bs, fft1k, and ludcmp), LINEARRANKTERM tool can prove only one example (bs) and it takes 0.59 seconds.

Our approach shows a promising result; even by a prototype implementation with a simple heuristic for atomic transition predicate generation, our tool outperforms other tools both in efficiency and effectiveness.

5 Related Work

Our work is inspired by the recent success of the algorithmic-learning-based approach to loop invariant inference [19–21]. In those papers, the problem of loop invariant generation is formulated as a problem of inferring an unknown quantifier-free formula. With a simple randomized mechanical teacher, a learning algorithm is adopted to infer an invariant for the given annotated loop. Instead of the costly fixpoint iteration, the learning algorithm revises its purported invariants by counterexamples from the teacher. The randomized teacher can guide the learning algorithm to find a loop invariant very efficiently since there are usually sufficiently many loop invariants.

TERMINATOR [13] is the most prominent termination analyzer which is successfully applied to an industrial practice [1]. Using transition invariants [25], TERMINATOR decomposes a termination problem of complex loops into easier ones. However, as reported in [13], the initial approach reveals that most of the analysis time is spent in reachability analysis that is to check if the current transition invariant reached a fixpoint.

Our work shares the same goal as several techniques [2, 8, 22, 27] which aims to improve the performance of the initial approach. To compute fixpoints efficiently, Berdine et al. [2] and Chawdhary et al. [8] use abstract interpretation [15]. We use in experiments LINEARRANKTERM [8] which adopts a new abstract domain tailored for termination proof. The new abstract domain is effective to prove the termination in most of the practical examples, but it simply gives up when a transition invariant of a loop is beyond its expressivity. Kroening et al. [22] and

³ The original benchmark suite can be also found at

<http://archi.snu.ac.kr/realtimetime/benchmark/>

Tsitovich et al. [27] use compositional transition invariants. Compositional transition invariants are the ones that are closed under composition with themselves. If a transition invariant covers several iterations of a loop and is compositional, it covers the entire iterations. Since compositional transition invariants can be found by considering only several iterations, they are sometimes discovered earlier than the one that covers the entire iterations. However, not all terminating programs have a compositional transition invariant.

Our technique can be easily extended with more sophisticated ranking function synthesis algorithms, such as lexicographic linear ranking functions [4] or bit-vector relations [11]. In this paper we use the ranking function synthesis algorithm for simple linear loops [24], which has been proven to be effective on realistic programs.

6 Conclusion

In this paper, we present an algorithmic-learning-based termination analysis technique. By combining transition predicate abstraction, algorithmic learning, and decision procedures, the technique can efficiently compute transition invariants as proofs of program termination. Compared to the previous approaches, our technique does not commit to any particular one, thus can prove the termination of the examples that previous techniques simply give up and report possible non-termination. We compare our technique with others on several benchmarks from literature. The result shows that the new technique outperforms the others both in efficiency and effectiveness.

Although our heuristic for selecting initial atomic transition predicates is effective, a complete predicate synthesis technique will be useful. Extending our learning-based framework to support more features such as function calls and pointers is certainly desirable. Several optimizations under the learning-based framework are to be explored. A more powerful well-foundedness checker should make the framework even more effective. An incremental learning algorithm for Boolean functions [9] should improve the efficiency of our technique as well.

Acknowledgement. We would like to thank anonymous referees for their comments and appreciations. We are grateful to Aziem Chawdhary, Peter O’Hearn, and Hongseok Yang for letting us use Windows device drivers examples. Especially, Aziem helps us a lot on the comparison with LINEAR RANK TERM analyzer. We also thank to Hakjoo Oh, Daejun Park, Yungbum Jung, Deokhwan Kim, and Sungkeun Cho for their comments.

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys (2006)

2. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O’Hearn, P.: Variance analyses from invariance analyses. In: POPL (2007)
3. Berdine, J., Cook, B., Distefano, D., O’Hearn, P.W.: Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear Ranking with Reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
5. 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)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of Polynomial Programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 113–129. Springer, Heidelberg (2005)
7. Bshouty, N.H.: Exact learning Boolean function via the monotone theory. Information and Computation 123(1), 146–153 (1995)
8. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking Abstractions. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 148–162. Springer, Heidelberg (2008)
9. Chen, Y.F., Wang, B.Y.: Learning Boolean Functions Incrementally. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 55–70. Springer, Heidelberg (2012)
10. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving Conditional Termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008)
11. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking Function Synthesis for Bit-Vector Relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)
12. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
13. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
14. Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: PLDI (2007)
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
16. 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)
17. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for Termination. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 304–319. Springer, Heidelberg (2010)
18. Heizmann, M., Jones, N.D., Podelski, A.: Size-Change Termination and Transition Invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 22–50. Springer, Heidelberg (2010)
19. Jung, Y., Kong, S., Wang, B.-Y., Yi, K.: Deriving Invariants by Algorithmic Learning, Decision Procedures, and Predicate Abstraction. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 180–196. Springer, Heidelberg (2010)

20. Jung, Y., Lee, W., Wang, B.-Y., Yi, K.: Predicate Generation for Learning-Based Quantifier-Free Loop Invariant Inference. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 205–219. Springer, Heidelberg (2011)
21. 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)
22. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination Analysis with Compositional Transition Invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
23. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
24. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
25. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS (2004)
26. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL (2005)
27. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop Summarization and Termination Analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)

Automated Termination Proofs for Java Programs with Cyclic Data*

Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. In earlier work, we developed a technique to prove termination of Java programs automatically: first, Java programs are automatically transformed to term rewrite systems (TRSs) and then, existing methods and tools are used to prove termination of the resulting TRSs. In this paper, we extend our technique in order to prove termination of algorithms on cyclic data such as cyclic lists or graphs automatically. We implemented our technique in the tool AProVE and performed extensive experiments to evaluate its practical applicability.

1 Introduction

Techniques to prove termination automatically are essential in program verification. While approaches and tools for automated termination analysis of *term rewrite systems* (TRSs) and of *logic programs* have been studied for decades, in the last years the focus has shifted toward imperative languages like C or Java.

Most techniques for imperative languages prove termination by synthesizing ranking functions (e.g., [12][26]) and localize the termination test using Ramsey's theorem [23][27]. Such techniques are for instance used in the tools Terminator [4][13] and LoopFrog [22][31] to analyze termination of C programs. To handle the heap, one can use an abstraction [14] to integers based on separation logic [24].

On the other hand, there also exist *transformational approaches* which automatically transform imperative programs to TRSs or to logic programs. They allow to re-use the existing techniques and tools from term rewriting or logic programming also for imperative programs. In [17], C is analyzed by a transformation to TRSs and the tools Julia [30] and COSTA [2] prove termination of Java via a transformation to constraint logic programs. To deal with the heap, they also use an abstraction to integers and represent objects by their *path length*.

In [6][7][8][25] we presented an alternative approach for termination of Java via a transformation to TRSs. Like [2][30], we consider Java Bytecode (JBC) to avoid dealing with all language constructs of Java. This is no restriction, since Java compilers automatically translate Java to JBC. Indeed, our implementation handles the Java Bytecode produced by Oracle's standard compiler. In contrast to other approaches, we do not treat the heap by an abstraction to integers, but by an abstraction to *terms*. So for any class C1 with n non-static fields, we use an n -ary function symbol C1. For example, consider a class List with two fields value and next. Then every List object is encoded as a term List(v, n) where

* Supported by the DFG grant GI 274/5-3.

v is the value of the current element and n is the encoding of the next element. Hence, a list “[1, 2]” is encoded by the term `List(1, List(2, null))`. In this way, our encoding maintains much more information from the original program than a (fixed) abstraction to integers. Now the advantage is that for any algorithm, existing tools from term rewriting can automatically search for (possibly different) suitable well-founded orders comparing arbitrary forms of terms. For more information on techniques for termination analysis of term rewriting, see, e.g., [16, 20, 33]. As shown in the annual *International Termination Competition*,¹ due to this flexibility, the implementation of our approach in the tool AProVE [19] is currently the most powerful termination prover for Java.

In this paper, we extend our technique to handle algorithms whose termination depends on cyclic objects (e.g., lists like “[0, 1, 2, 1, 2, …]” or cyclic graphs). Up to now, transformational approaches could not deal with such programs. Similar to related approaches based on separation logic [45, 10, 11, 28, 32], our technique relies on suitable predicates describing properties of the heap. Like [28], but in contrast to several previous works, our technique derives these heap predicates *automatically* from the input program and it works automatically for arbitrary data structures (i.e., not only for lists). We integrated this new technique in our fully automated termination analysis and made the resulting termination tool available via a web interface [1]. This tool automatically proves termination of Java programs on possibly cyclic data, i.e., the user does not have to provide loop preconditions, invariants, annotations, or any other manual pre-processing.

Our technique works in two steps: first, a JBC program is transformed into a *termination graph*, which is a finite representation of all program runs. This graph takes all sharing effects into account. Afterwards, a TRS is generated from the graph. In a similar way, we also developed techniques to analyze termination of other languages like Haskell [21] or Prolog [29] via a translation to TRSs.

Of course, one could also transform termination graphs into other formalisms than TRSs. For example, by fixing the translation from objects to integers, one could easily generate integer transition systems from the termination graph. Then the contributions of the current paper can be used as a general pre-processing approach to handle cyclic objects, which could be coupled with other termination tools. However, for methods whose termination does *not* rely on cyclic data, our technique is able to transform data objects into terms. For such methods, the power of existing tools for TRSs allows us to find more complex termination arguments automatically. By integrating the contributions of the current paper into our TRS-based framework, the resulting tool combines the new approach for cyclic data with the existing TRS-based approach for non-cyclic data.

In Sect. 2.1, we consider three typical classes of algorithms which rely on data that could be cyclic. The first class are algorithms where the cyclicity is *irrelevant* for termination. So for termination, one only has to inspect a non-cyclic part of the objects. For example, consider a doubly-linked list where the predecessor of the first and the successor of the last element are `null`. Here, a traversal only following the `next` field obviously terminates. To handle such algorithms,

¹ See http://termination-portal.org/wiki/Termination_Competition

in Sect. 2 we recapitulate our termination graph framework and present a new improvement to detect irrelevant cyclicity automatically.

The second class are algorithms that mark every visited element in a cyclic object and terminate when reaching an already marked element. In Sect. 3, we develop a technique based on SMT solving to detect such *marking algorithms* by analyzing the termination graph and to prove their termination automatically.

The third class are algorithms that terminate because an element in a cyclic object is guaranteed to be visited a second time (i.e., the algorithms terminate when reaching a specified sentinel element). In Sect. 4, we extend termination graphs by representing *definite* sharing effects. Thus, we can now express that by following some field of an object, one eventually reaches another specific object. In this way, we can also prove termination of well-known algorithms like the in-place reversal for pan-handle lists [10] automatically.

We implemented all our contributions in the tool AProVE. Sect. 5 shows their applicability by an evaluation on a large benchmark collection (including numerous standard Java library programs, many of which operate on cyclic data). In our experiments, we observed that the three considered classes of algorithms capture a large portion of typical programs on cyclic data. For the treatment of (general classes of) other programs, we refer to our earlier papers [6, 7, 25]. Moreover, in [8] we presented a technique that uses termination graphs to also detect non-termination. By integrating the new contributions of the current paper into our approach, our tool can now automatically prove termination for programs that contain methods operating on cyclic data as well as other methods operating on non-cyclic data. For the proofs of the theorems as well as all formal definitions needed for the construction of termination graphs, we refer to [9].

2 Handling Irrelevant Cycles

We restrict ourselves to programs without method calls, arrays, exception handlers, static fields, floating point numbers, class initializers, reflection, and multithreading to ease the presentation. However, our implementation supports these features, except reflection and multithreading. For further details, see [6, 7, 8].

```
class L1 {
    L1 p, n;
    static int length(L1 x) {
        int r = 1;
        while (null != (x = x.n)) {
            r++;
        }
        return r;
    }
}
```

Fig. 1. Java Program

In Fig. 1, L1 is a class for (doubly-linked) lists where n and p point to the next and previous element. For brevity, we omitted a field for the value of elements. The

```
00:  iconst_1      #load 1
01:  istore_1      #store to r
02:  aconst_null   #load null
03:  aload_0        #load x
04:  getfield n    #get n from x
07:  dup            #duplicate n
08:  astore_0        #store to x
09:  if_acmpeq 18  #jump if
                  # x.n == null
12:  iinc 1, 1     #increment r
15:  goto 02
18:  iload_1        #load r
19:  ireturn         #return r
```

Fig. 2. JBC for length

method `length` initializes a variable `r` for the result and traverses the list until `x` is `null`. Fig. 2 shows the corresponding JBC obtained by the Java compiler.

After introducing program states in Sect. 2.1, we explain how termination graphs are generated in Sect. 2.2. Sect. 2.3 shows the transformation from termination graphs to TRSs. While this two-step transformation was already presented in our earlier papers, here we extend it by an improved handling of cyclic objects in order to prove termination of algorithms like `length` automatically.

2.1 Abstract States in Termination Graphs

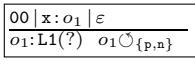
 We generate a graph of abstract states from $\text{STATES} = \text{PPOS} \times \text{LOCVAR} \times \text{OPSTACK} \times \text{HEAP} \times \text{ANNOTATIONS}$, where PPOS

Fig. 3. State A is the set of all program positions. Fig. 3 depicts the initial state for the method `length`. The first three components of a state are in the first line, separated by “|”. The first component is the program position, indicated by the index of the next instruction. The second component represents the local variables as a list of references, i.e., $\text{LOCVAR} = \text{REFS}^*$ ². To ease readability, in examples we denote local variables by names instead of numbers. So “`x: o1`” indicates that the 0-th local variable `x` has the value `o1`. The third component is the operand stack $\text{OPSTACK} = \text{REFS}^*$ for temporary results of JBC instructions. The empty stack is denoted by ε and “`o1, o2`” is a stack with top element `o1`.

Below the first line, information about the heap is given by a function from $\text{HEAP} = \text{REFS} \rightarrow \text{INTS} \cup \text{UNKNOWN} \cup \text{INSTANCES} \cup \{\text{null}\}$ and by a set of annotations specifying sharing effects in parts of the heap that are not explicitly represented. For integers, we abstract from the different types of bounded integers in Java and consider unbounded integers instead, i.e., we cannot handle problems related to overflows. We represent unknown integers by intervals, i.e., $\text{INTS} = \{\{x \in \mathbb{Z} \mid a \leq x \leq b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}$. For readability, we abbreviate intervals such as $(-\infty, \infty)$ by \mathbb{Z} and $[1, \infty)$ by $[>0]$.

Let `CLASSNAMES` contain all classes and interfaces in the program. The values $\text{UNKNOWN} = \text{CLASSNAMES} \times \{?\}$ denote that a reference points to an unknown object or to `null`. Thus, “`o1:L1(?)`” means that at address `o1`, we have an instance of `L1` (or of its subclasses) with unknown field values or that `o1` is `null`.

To represent actual objects, we use $\text{INSTANCES} = \text{CLASSNAMES} \times (\text{FIELDIDS} \rightarrow \text{REFS})$, where `FIELDIDS` is the set of all field identifiers. To prevent ambiguities, in general the `FIELDIDS` also contain the respective class names. Thus, “`o2:L1(p = o3, n = o4)`” means that at address `o2`, we have some object of type `L1` whose field `p` contains the reference `o3` and whose field `n` contains `o4`.

² To avoid a special treatment of integers (which are primitive values in JBC), we also represent them using references to the heap.

In our representation, if a state contains the references o_1 and o_2 , then the objects reachable from o_1 resp. o_2 are disjoint³ and tree-shaped (and thus acyclic), unless explicitly stated otherwise. This is orthogonal to the default assumptions in separation logic, where sharing is allowed unless stated otherwise, cf. e.g. [32]. In our states, one can either express sharing directly (e.g., “ $o_1 : \text{L1}(p = o_2, n = o_1)$ ” implies that o_1 reaches o_2 and is cyclic) or use *annotations* to indicate (possible) sharing in parts of the heap that are not explicitly represented.

The first kind of annotation is the *equality annotation* $o =? o'$, meaning that o and o' could be the same. We only use this annotation if $h(o) \in \text{UNKNOWN}$ or $h(o') \in \text{UNKNOWN}$, where h is the heap of the state. The second annotation is the *joinability annotation* $o \swarrow o'$, meaning that o and o' possibly have a common successor. To make this precise, let $o_1 \xrightarrow{f} o_2$ denote that the object at o_1 has a field $f \in \text{FIELDIDs}$ with o_2 as its value (i.e., $h(o_1) = (\text{C1}, e) \in \text{INSTANCES}$ and $e(f) = o_2$). For any $\pi = f_1 \dots f_n \in \text{FIELDIDs}^*$, $o_1 \xrightarrow{\pi} o_{n+1}$ denotes that there exist o_2, \dots, o_n with $o_1 \xrightarrow{f_1} o_2 \xrightarrow{f_2} \dots \xrightarrow{f_{n-1}} o_n \xrightarrow{f_n} o_{n+1}$. Moreover, $o_1 \xrightarrow{\varepsilon} o'_1$ iff $o_1 = o'_1$. Then $o \swarrow o'$ means that there could be some o'' and some π and τ such that $o \xrightarrow{\pi} o'' \xleftarrow{\tau} o'$, where $\pi \neq \varepsilon$ or $\tau \neq \varepsilon$.

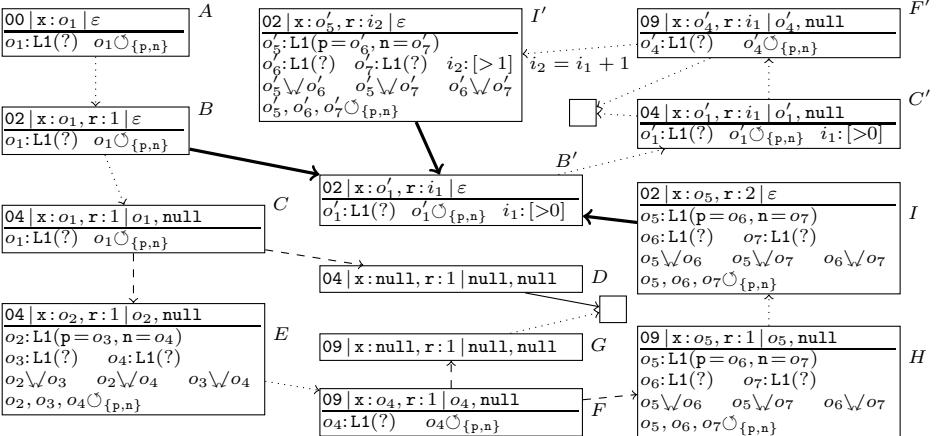
In our earlier papers [6, 25] we had another annotation to denote references that may point to non-tree-shaped objects. In the translation to terms later on, all these objects were replaced by fresh variables. But in this way, one cannot prove termination of `length`. To maintain more information about possibly non-tree-shaped objects, we now introduce two new *shape annotations* $o\diamond$ and $o\circlearrowright_{FI}$ instead. The *non-tree annotation* $o\diamond$ means that o might be not tree-shaped. More precisely, there could be a reference o' with $o \xrightarrow{\pi_1} o'$ and $o \xrightarrow{\pi_2} o'$ where π_1 is no prefix of π_2 and π_2 is no prefix of π_1 . However, these two paths from o to o' may not traverse any cycles (i.e., there are no prefixes τ_1, τ_2 of π_1 or of π_2 where $\tau_1 \neq \tau_2$, but $o \xrightarrow{\tau_1} o''$ and $o \xrightarrow{\tau_2} o''$ for some o''). The *cyclicity annotation* $o\circlearrowright_{FI}$ means that there could be cycles including o or reachable from o . However, any cycle must use at least the fields in $FI \subseteq \text{FIELDIDs}$. In other words, if $o \xrightarrow{\tau} o' \xrightarrow{\tau} o'$ for some $\tau \neq \varepsilon$, then τ must contain all fields from FI . We often write \circlearrowright instead of $\circlearrowright_\emptyset$. Thus in Fig. 3, $o_1\circlearrowright_{\{p,n\}}$ means that there may be cycles reachable from o_1 and that any such cycle contains at least one `p` and one `n` field.

2.2 Constructing the Termination Graph

Our goal is to prove termination of `length` for all doubly-linked lists without “real” cycles (i.e., there is no cycle traversing only `n` or only `p` fields). Hence, A is the initial state when calling the method with such an input list.⁴ From A , the termination graph in Fig. 4 is constructed by symbolic evaluation. First, `iconst_1` loads the constant 1 on the operand stack. This leads to a new state connected to A by an *evaluation edge* (we omitted this state from Fig. 4 for

³ An exception are references to `null` or `INTS`, since in JBC, integers are primitive values where one cannot have any side effects. So if h is the heap of a state and $h(o_1) = h(o_2) \in \text{INTS}$ or $h(o_1) = h(o_2) = \text{null}$, then one can always assume $o_1 = o_2$.

⁴ The state A is obtained automatically when generating the termination graph for a program where `length` is called with an arbitrary such input list, cf. Sect. 5.

Fig. 4. Termination Graph for `length`

reasons of space). Then `istore_1` stores the constant 1 from the top of the operand stack in the first local variable `r`. In this way, we obtain state *B* (in Fig. 4 we use dotted edges to indicate several steps). Formally, the constant 1 is represented by some reference $i \in \text{REFS}$ that is mapped to $[1, 1] \in \text{INTS}$ by the heap. However, we shortened this for the presentation and just wrote $r : 1$.

In *B*, we load `null` and the value of `x` (i.e., o_1) on the operand stack, resulting in *C*. In *C*, the result of `getfield` depends on the value of o_1 . Hence, we perform a case analysis (a so-called *instance refinement*) to distinguish between the possible types of o_1 (and the case where o_1 is `null`). So we obtain *D* where o_1 is `null`, and *E* where o_1 points to an actual object of type `L1`. To get single static assignments, we rename o_1 to o_2 in *E* and create fresh references o_3 and o_4 for its fields `p` and `n`. We connect *D* and *E* by dashed *refinement edges* to *C*.

In *E*, our annotations have to be updated. If o_1 can reach a cycle, then this could also hold for its successors. Thus, we copy $\circlearrowleft_{\{p,n\}}$ to the newly-created successors o_3 and o_4 . Moreover, if o_2 (o_1 under its new name) can reach itself, then its successors might also reach o_2 and they might also reach each other. Thus, we create \swarrow annotations indicating that each of these references may share with any of the others. We do not have to create any equality annotations. The annotation $o_2 =? o_3$ (and $o_2 =? o_4$) is not needed because if the two were equal, they would form a cycle involving only one field, which contradicts $\circlearrowleft_{\{p,n\}}$. Furthermore, we do not need $o_3 =? o_4$, as o_1 was not marked with \diamond .

D ends the program (by an exception), indicated by an empty box. In *E*, `getfield n` replaces o_2 on the operand stack by the value o_4 of its field `n`, `dup` duplicates the entry o_4 on the stack, and `astore_0` stores one of these entries in `x`, resulting in *F*. We removed o_2 and o_3 which are no longer used in local variables or the operand stack. To evaluate `if_acmpneq` in *F*, we branch depending on the equality of the two top references on the stack. So we need an *instance refinement* and create *G* where o_4 is `null`, and *H* where o_4 refers to an actual object. The annotations in *H* are constructed from *F* just as *E* was constructed from *C*.

G results in a program end. In H , \mathbf{r} 's value is incremented to 2 and we jump back to instruction 02, resulting in I . We could continue symbolic evaluation, but this would not yield a finite termination graph. Whenever two states like B and I are at the same program position, we use *generalization* (or *widening* [14]) to find a common representative B' of both B and I . By suitable heuristics, our automation ensures that one always reaches a finite termination graph after finitely many generalization steps [8]. The values for references in B' include all values that were possible in B or I . Since \mathbf{r} had the value 1 in B and 2 in I , this is generalized to the interval $[>0]$ in B' . Similarly, since \mathbf{x} was UNKNOWN in B but a non-null list in I , this is generalized to an UNKNOWN value in B' .

We draw *instance edges* (depicted by thick arrows) from B and I to B' , indicating that all concrete (i.e., non-abstract) program states represented by B or I are also represented by B' . So B and I are *instances* of B' (written $B \sqsubseteq B'$, $I \sqsubseteq B'$) and any evaluation starting in B or I could start in B' as well.

From B' on, symbolic evaluation yields analogous states as when starting in B . The only difference is that now, \mathbf{r} 's value is an unknown positive integer. Thus, we reach I' , where \mathbf{r} 's value i_2 is the incremented value of i_1 and the edge from F' to I' is labeled with “ $i_2 = i_1 + 1$ ” to indicate this relation. Such labels are used in Sect. 2.3 when generating TRSs from termination graphs. The state I' is similar to I , and it is again represented by B' . Thus, we can draw an instance edge from I' to B' to “close” the graph, leaving only program ends as leaves.

A sequence of concrete states c_1, c_2, \dots is a *computation path* if c_{i+1} is obtained from c_i by standard JBC evaluation. A computation sequence is *represented* by a termination graph if there is a path $s_1^1, \dots, s_1^{k_1}, s_2^1, \dots, s_2^{k_2}, \dots$ of states in the termination graph such that $c_i \sqsubseteq s_i^1, \dots, c_i \sqsubseteq s_i^{k_i}$ for all i and such that all labels on the edges of the path (e.g., “ $i_2 = i_1 + 1$ ”) are satisfied by the corresponding values in the concrete states. Thm. I shows that if a concrete state c_1 is an instance of some state s_1 in the termination graph, then every computation path starting in c_1 is represented by the termination graph. Thus, every infinite computation path starting in c_1 corresponds to a cycle in the termination graph.

Theorem 1 (Soundness of Termination Graphs). *Let G be a termination graph, s_1 some state in G , and c_1 some concrete state with $c_1 \sqsubseteq s_1$. Then any computation sequence c_1, c_2, \dots is represented by G .*

2.3 Proving Termination via Term Rewriting

From the termination graph, one can generate a TRS with built-in integers [18] that only terminates if the original program terminates. To this end, in [25] we showed how to encode each state of a termination graph as a term and each edge as a rewrite rule. We now extend this encoding to the new annotations \Diamond and \heartsuit in such a way that one can prove termination of algorithms like `length`.

To encode states, we convert the values of local variables and operand stack entries to terms. References with unknown value are converted to variables of the same name. So the reference i_1 in state B' is converted to the variable i_1 .

The `null` reference is converted to the constant `null` and for objects, we use the name of their class as a function symbol. The arguments of that function

correspond to the fields of the class. So a list \mathbf{x} of type $\mathbf{L1}$ where $\mathbf{x.p}$ and $\mathbf{x.n}$ are `null` would be converted to the term $\mathbf{L1}(\mathbf{null}, \mathbf{null})$ and \mathbf{o}_2 from state E would be converted to the term $\mathbf{L1}(o_3, o_4)$ if it were not possibly cyclic.

In [25], we had to exclude objects that were not tree-shaped from this translation. Instead, accesses to such objects always yielded a fresh, unknown variable. To handle objects annotated with \diamond , we now use a simple unrolling when transforming them to terms. Whenever a reference is changed in the termination graph, then all its occurrences in the unrolled term are changed simultaneously in the corresponding TRS. To handle the annotation \circ_{FI} , now we only encode a *subset* of the fields of each class when transforming objects to terms. This subset is chosen such that at least one field of FI is disregarded in the term encoding.⁵ Hence, when only regarding the encoded fields, the data objects are acyclic and can be represented as terms. To determine which fields to drop from the encoding, we use a heuristic which tries to disregard fields without read access.

In our example, all cyclicity annotations have the form $\circ_{\{\mathbf{p}, \mathbf{n}\}}$ and \mathbf{p} is never read. Hence, we only consider the field \mathbf{n} when encoding $\mathbf{L1}$ -objects to terms. Thus, \mathbf{o}_2 from state E would be encoded as $\mathbf{L1}(o_4)$. Now any read access to \mathbf{p} would have to be encoded as returning a fresh variable.

For every state we use a function with one argument for each local variable and each entry of the operand stack. So E is converted to $f_E(\mathbf{L1}(o_4), 1, \mathbf{L1}(o_4), \mathbf{null})$.

To encode the edges of the termination graph as rules, we consider the different kinds of edges. For a chain of *evaluation edges*, we obtain a rule whose left-hand side is the term resulting from the first state and whose right-hand side results from the last state of the chain. So the edges from E to F result in

$$f_E(\mathbf{L1}(o_4), 1, \mathbf{L1}(o_4), \mathbf{null}) \rightarrow f_F(o_4, 1, o_4, \mathbf{null}).$$

In term rewriting [3], a rule $\ell \rightarrow r$ can be applied to a term t if there is a substitution σ with $\ell\sigma = t'$ for some subterm t' of t . The application of the rule results in a variant of t where t' is replaced by $r\sigma$. For example, consider a concrete state where \mathbf{x} is a list of length 2 and the program counter is 04. This state would be an instance of the abstract state E and it would be encoded by the term $f_E(\mathbf{L1}(\mathbf{L1}(\mathbf{null})), 1, \mathbf{L1}(\mathbf{L1}(\mathbf{null})), \mathbf{null})$. Now applying the rewrite rule above yields $f_F(\mathbf{L1}(\mathbf{null}), 1, \mathbf{L1}(\mathbf{null}), \mathbf{null})$. In this rule, we can see the main termination argument: Between E and F , one list element is “removed” and the list has finite length (when only regarding the \mathbf{n} field). A similar rule is created for the evaluations that lead to state F' , where all occurrences of 1 are replaced by i_1 .

In our old approach [25], the edges from E to F would result in $f_E(\mathbf{L1}(o_4), 1, \mathbf{L1}(o_4), \mathbf{null}) \rightarrow f_F(o'_4, 1, o'_4, \mathbf{null})$. Its right-hand side uses the fresh variable o'_4 instead of o_4 , since this was the only way to represent cyclic objects in [25]. Since o'_4 could be instantiated by any term during rewriting, this TRS is not terminating.

For *refinement edges*, we use the term for the target state on both sides of the resulting rule. However, on the left-hand side, we label the outermost function

⁵ Of course, if $FI = \emptyset$, then we still handle cyclic objects as before and represent any access to them by a fresh variable.

symbol with the source state. So for the edge from F to H , we have the term for H on both sides of the rule, but on the left-hand side we replace f_H by f_F :

$$f_F(L1(o_7), 1, L1(o_7), \text{null}) \rightarrow f_H(L1(o_7), 1, L1(o_7), \text{null})$$

For *instance edges*, we use the term for the source state on both sides of the resulting rule. However, on the right-hand side, we label the outermost function with the target state instead. So for the edge from I to B' , we have the term for I on both sides of the rule, but on the right-hand side we replace f_I by $f_{B'}$:

$$f_I(L1(o_7), 2) \rightarrow f_{B'}(L1(o_7), 2)$$

For termination, it suffices to convert just the (non-trivial) SCCs of the termination graph to TRSs. If we do this for the only SCC B', \dots, I', \dots, B' of our graph, and then “merge” rewrite rules that can only be applied after each other [25], then we obtain one rule encoding the only possible way through the loop:

$$f_{B'}(L1(L1(o_7)), i_1) \rightarrow f_{B'}(L1(o_7), i_1 + 1)$$

Here, we used the information on the edges from F' to I' to replace i_2 by $i_1 + 1$. Termination of this rule is easily shown automatically by termination provers like AProVE, although the original Java program worked on cyclic objects. However, our approach automatically detects that the objects are not cyclic anymore if one uses a suitable projection that only regards certain fields of the objects.

Theorem 2 (Proving Termination of Java by TRSs). *If the TRSs resulting from the SCCs of a termination graph G are terminating, then G does not represent any infinite computation sequence. So by Thm. 1, the original JBC program is terminating for all concrete states c where $c \sqsubseteq s$ for some state s in G .*

3 Handling Marking Algorithms on Cyclic Data

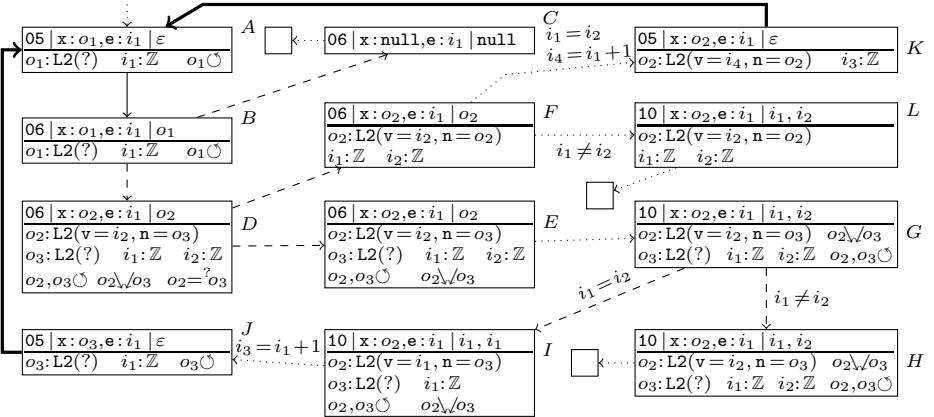
```
public class L2 {
    int v;
    L2 n;
    static void visit(L2 x){
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n; }}
```

Fig. 5. Java Program

We now regard lists with a “next” field n where every element has an integer value v . The method `visit` stores the value of the first list element. Then it iterates over the list elements as long as they have the same value and “marks” them by modifying their value. If

```
00: aload_0      #load x
01: getfield v   #get v from x
04: istore_1     #store to e
05: aload_0      #load x
06: getfield v   #get v from x
09: iload_1      #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0      #load x
14: iload_1      #load e
15: iconst_1     #load 1
16: iadd         #add e and 1
17: putfield v   #store to x.v
20: aload_0      #load x
21: getfield n   #get n from x
24: astore_0     #store to x
25: goto 5
28: return
```

Fig. 6. JBC for `visit`

**Fig. 7.** Termination Graph for visit

all list elements had the same value initially, then the iteration either ends with a `NullPointerException` (if the list is non-cyclic) or because some element is visited for the second time (this is detected by its modified “marked” value).⁶ We illustrate the termination graph of `visit` in Sect. 3.1 and extend our approach in order to prove termination of such marking algorithms in Sect. 3.2.

3.1 Constructing the Termination Graph

When calling `visit` for an arbitrary (possibly cyclic) list, one reaches state *A* in Fig. 7 after one loop iteration by symbolic evaluation and generalization. Now `aload_0` loads the value o_1 of `x` on the operand stack, yielding state *B*.

To evaluate `getfield v`, we perform an instance refinement and create a successor *C* where o_1 is `null` and a successor *D* where o_1 is an actual instance of `L2`. As in Fig. 4, we copy the cyclicity annotation to o_3 and allow o_2 and o_3 to join. Furthermore, we add $o_2 = ? o_3$, since o_2 could be a cyclic one-element list.

In *C*, we end with a `NullPointerException`. Before accessing o_2 ’s fields, we have to resolve all possible equalities. We obtain *E* and *F* by an *equality refinement*, corresponding to the cases $o_2 \neq o_3$ and $o_2 = o_3$. *F* needs no annotations anymore, as all reachable objects are completely represented in the state.

In *E* we evaluate `getfield`, retrieving the value i_2 of the field `v`. Then we load `e`’s value i_1 on the operand stack, which yields *G*. To evaluate `if_icmpne`, we branch depending on the inequality of the top stack entries i_1 and i_2 , resulting in *H* and *I*. We label the refinement edges with the respective integer relations.

In *I*, we add 1 to i_1 , creating i_3 , which is written into the field `v` of o_2 . Then, the field `n` of o_2 is retrieved, and the obtained reference o_3 is written into `x`, leading to *J*. As *J* is a renaming of *A*, we draw an instance edge from *J* to *A*.

⁶ While termination of `visit` can also be shown by the technique of Sect. 4 which detects whether an element is visited twice, the technique of Sect. 4 fails for analogous marking algorithms on graphs which are easy to handle by the approach of Sect. 3, cf. Sect. 5. So the techniques of Sect. 3 and 4 do not subsume each other.

The states following F are analogous, i.e., when reaching `if_icmpne`, we create successors depending on whether $i_1 = i_2$. In that case, we reach K , where we have written the new value $i_4 = i_1 + 1$ into the field `v` of o_2 . Since K is also an instance of A , this concludes the construction of the termination graph.

3.2 Proving Termination of Marking Algorithms

To prove termination of algorithms like `visit`, we try to find a suitable *marking property* $M \subseteq \text{REFS} \times \text{STATES}$. For every state s with heap h , we have $(o, s) \in M$ if o is reachable⁷ in s and if $h(o)$ is an object satisfying a certain property. We add a local variable named c_M to each state which counts the number of references in M . More precisely, for each concrete state s with “ $c_M : i$ ” (i.e., the value of the new variable is the reference i), $h(i) \in \text{INTS}$ is the singleton set containing the number of references o with $(o, s) \in M$. For any abstract state s with “ $c_M : i$ ” that represents some concrete state s' (i.e., $s' \sqsubseteq s$), the interval $h(i)$ must contain an upper bound for the number of references o with $(o, s') \in M$.

In our example, we consider the property $\text{L2.v} = i_1$, i.e., c_M counts the references to L2-objects whose field `v` has value i_1 . As the loop in `visit` only continues if there is such an object, we have $c_M > 0$. Moreover, in each iteration, the field `v` of some L2-object is set to a value i_3 resp. i_4 which is *different* from i_1 . Thus, c_M decreases. We now show how to find this termination proof automatically.

To detect a suitable marking property automatically, we restrict ourselves to properties “ $\text{C1.f} \bowtie i$ ”, where C1 is a class, f a field in C1 , i a (possibly unknown) integer, and \bowtie an integer relation. Then $(o, s) \in M$ iff $h(o)$ is an object of type C1 (or a subtype of C1) whose field f stands in relation \bowtie to the value i .

The first step is to find some integer reference i that is never changed in the SCC. In our example, we can easily infer this for i_1 automatically.⁸

The second step is to find C1 , f , and \bowtie such that every cycle of the SCC contains some state where $c_M > 0$. We consider those states whose incoming edge has a label “ $i \bowtie \dots$ ” or “ $\dots \bowtie i$ ”. In our example, I ’s incoming edge is labeled with “ $i_1 = i_2$ ” and when comparing i_1 and i_2 in G , i_2 was the value of o_2 ’s field `v`, where o_2 is an L2-object. This suggests the marking property “ $\text{L2.v} = i_1$ ”. Thus, c_M now counts the references to L2-objects whose field `v` has the value i_1 . So the cycle A, \dots, E, \dots, A contains the state I with $c_M > 0$ and one can automatically detect that A, \dots, F, \dots, A has a similar state with $c_M > 0$.

In the third step, we add c_M as a new local variable to all states of the SCC. For instance, in A to G , we add “ $c_M : i$ ” to the local variables and “ $i : [\geq 0]$ ” to the knowledge about the heap. The edge from G to I is labeled with “ $i > 0$ ” (this will be used in the resulting TRS), and in I we know “ $i : [> 0]$ ”. It remains to explain how to detect changes of c_M . To this end, we use SMT solving.

A counter for “ $\text{C1.f} \bowtie i$ ” can only change when a new object of type C1 (or a subtype) is created or when the field C1.f is modified. So whenever “`new C1`”

⁷ Here, a reference o is *reachable* in a state s if s has a local variable or an operand stack entry o' such that $o' \xrightarrow{\pi} o$ for some $\pi \in \text{FIELDIDS}^*$.

⁸ Due to our single static assignment syntax, this follows from the fact that at all instance edges, i_1 is matched to i_1 .

(or “**new C1’**” for some subtype **C1’**) is called, we have to consider the default value d for the field **C1.f**. If the underlying SMT solver can prove that $\neg d \bowtie i$ is a tautology, then c_M can remain unchanged. Otherwise, to ensure that c_M is an upper bound for the number of objects in M , c_M is incremented by 1.

If a **putfield** replaces the value u in **C1.f** by w , we have three cases:

- (i) If $u \bowtie i \wedge \neg w \bowtie i$ is a tautology, then c_M may be decremented by 1.
- (ii) If $u \bowtie i \leftrightarrow w \bowtie i$ is a tautology, then c_M remains the same.
- (iii) In the remaining cases, we increment c_M by 1.

In our example, between I and J one writes i_3 to the field **v** of o_2 . To find out how c_M changes from I to J , we create a formula containing all information on the edges in the path up to now (i.e., we collect this information by going backwards until we reach a state like A with more than one predecessor). This results in $i_1 = i_2 \wedge i_3 = i_1 + 1$. To detect whether we are in case (i) above, we check whether the information in the path implies $u \bowtie i \wedge \neg w \bowtie i$. In our example, the previous value u of $o_2.\text{v}$ is i_1 and the new value w is i_3 . Any SMT solver for integer arithmetic can easily prove that the resulting formula

$$i_1 = i_2 \wedge i_3 = i_1 + 1 \rightarrow i_1 = i_1 \wedge \neg i_3 = i_1$$

is a tautology (i.e., its negation is unsatisfiable). Thus, c_M is decremented by 1 in the step from I to J . Since in I , we had “ $c_M : i'$ ” with “ $i : [> 0]$ ”, in J we have “ $c_M : i'$ ” with “ $i' : [\geq 0]$ ”. Moreover, we label the edge from I to J with the relation “ $i' = i - 1$ ” which is used when generating a TRS from the termination graph. Similarly, one can also easily prove that c_M decreases between F and K . Thm. 3 shows that Thm. II still holds when states are extended by counters c_M .

Theorem 3 (Soundness of Termination Graphs with Counters for Marking Properties). *Let G be a termination graph, s_1 some state in G , c_1 some concrete state with $c_1 \sqsubseteq s_1$, and M some marking property. If we extend all concrete states c with heap h by an extra local variable “ $c_M : i$ ” such that $h(i) = \{|\{(o, c) \in M\}|\}$ and if we extend abstract states as described above, then any computation sequence c_1, c_2, \dots is represented by G .*

We generate TRSs from the termination graph as before. So by Thm. 2 and 3, termination of the TRSs still implies termination of the original Java program.

Since the new counter is an extra local variable, it results in an extra argument of the functions in the TRS. So for the cycle $A, \dots, E, \dots A$, after some “merging” of rules, we obtain the following TRS. Here, the first rule may only be applied under the condition $i > 0$. For $A, \dots, F, \dots A$ we obtain similar rules.

$$\begin{array}{ll} f_A(\dots, i, \dots) \rightarrow f_I(\dots, i, \dots) \mid i > 0 & f_I(\dots, i, \dots) \rightarrow f_J(\dots, i - 1, \dots) \\ f_J(\dots, i', \dots) \rightarrow f_A(\dots, i', \dots) & \end{array}$$

Termination of the resulting TRS can easily be shown automatically by standard tools from term rewriting, which proves termination of the method **visit**.

4 Handling Algorithms with Definite Cyclicity

```
public class L3 {
    L3 n;
    void iterate() {
        L3 x = this.n;
        while (x != this)
            x = x.n; }}
```

Fig. 8. Java Program

The method in Fig. 8 traverses a cyclic list until it reaches the start again. It only terminates if by following the `n` field, we reach `null` or the first element again. We illustrate `iterate`'s termination graph in Sect. 4.1 and introduce a new *definite reachability* annotation for such algorithms. Afterwards, Sect. 4.2 shows how to prove their termination.

```
00:  aload_0      #load this
01:  getfield n   #get n from this
04:  astore_1      #store to x
05:  aload_1      #load x
06:  aload_0      #load this
07:  if_acmpeq 18 #jump if x == this
10:  aload_1      #load x
11:  getfield n   #get n from x
14:  astore_1      #store x
15:  goto 05
18:  return
```

Fig. 9. JBC for `iterate`

Fig. 10 shows the termination graph when calling `iterate` with an arbitrary list whose first element is on a cycle.⁹ In contrast to marking algorithms like `visit` in Sect. 3, `iterate` does not terminate for other forms of cyclic lists. State *A* is reached after evaluating the first three instructions, where the value

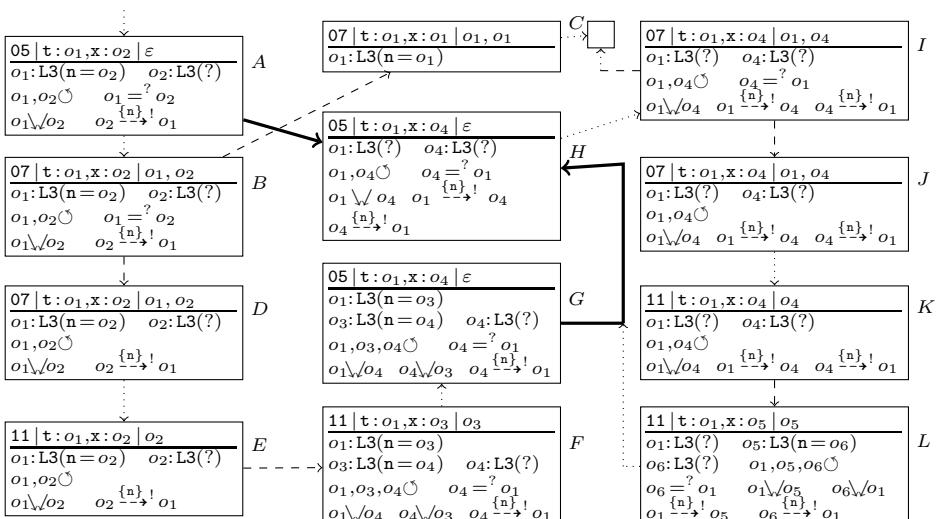


Fig. 10. Termination Graph for `iterate`

⁹ The initial state of `iterate`'s termination graph is obtained automatically when proving termination for a program where `iterate` is called with such lists, cf. Sect. 5.

o_2 of `this.n`¹⁰ is copied to `x`. In A , o_1 and o_2 are the first elements of the list, and $o_1 =? o_2$ allows that both are the same. Furthermore, both references are possibly cyclic and by $o_1 \setminus\! o_2$, o_2 may eventually reach o_1 again (i.e., $o_2 \xrightarrow{\pi} o_1$).

Moreover, we added a new annotation $o_2 \xrightarrow{\{n\}}! o_1$ to indicate that o_2 *definitely reaches* o_1 ¹¹. All previous annotations $=?$, $\setminus\!$, \diamond , \circlearrowleft extend the set of concrete states represented by an abstract state (by allowing more sharing). In contrast, a *definite reachability* annotation $o \xrightarrow{FI}! o'$ with $FI \subseteq \text{FIELDIDS}$ restricts the set of states represented by an abstract state. Now it only represents states where $o \xrightarrow{\pi} o'$ holds for some $\pi \in FI^*$. To ensure that the FI -path from o to o' is unique (up to cycles), FI must be *deterministic*. This means that for any class $C1$, FI contains at most one of the fields of $C1$ or its superclasses. Moreover, we only use $o \xrightarrow{FI}! o'$ if $h(o) \in \text{UNKNOWN}$ for the heap h of the state.

In A , we load the values o_2 and o_1 of `x` and `this` on the stack. To evaluate `if_acmpeq` in B , we need an equality refinement w.r.t. $o_1 =? o_2$. We create C for the case where $o_1 = o_2$ (which ends the program) and D for $o_1 \neq o_2$.

In D , we load `x`'s value o_2 on the stack again. To access its field `n` in E , we need an instance refinement for o_2 . By $o_2 \xrightarrow{\{n\}}! o_1$, o_2 's value is not `null`. So there is only one successor F where we replace o_2 by o_3 , pointing to an L3-object. The annotation $o_2 \xrightarrow{\{n\}}! o_1$ is moved to the value of the field `n`, yielding $o_4 \xrightarrow{\{n\}}! o_1$.

In F , the value o_4 of o_3 's field `n` is loaded on the stack and written to `x`. Then we jump back to instruction 05. As G and A are at the same program position, they are generalized to a new state H which represents both G and A . H also illustrates how definite reachability annotations are generated automatically: In A , `this` reaches `x` in one step, i.e., $o_1 \xrightarrow{n} o_2$. Similarly in G , `this` reaches `x` in two steps, i.e., $o_1 \xrightarrow{n n} o_4$. To generalize this connection between `this` and `x` in the new state H where “`this : o1`” and “`x : o4`”, one generates the annotation $o_1 \xrightarrow{\{n\}}! o_4$ in H . Thus, `this` definitely reaches `x` in arbitrary many steps.

From H , symbolic evaluation continues just as from A . So we reach the states I, J, K, L (corresponding to B, D, E, F , respectively). In L , the value o_6 of `x.n` is written to `x` and we jump back to instruction 05. There, o_5 is not referenced anymore. However, we had $o_1 \xrightarrow{\{n\}}! o_5$ in state L . When garbage collecting o_5 , we “transfer” this annotation to its `n`-successor o_6 , generating $o_1 \xrightarrow{\{n\}}! o_6$. Now the resulting state is just a variable renaming of H , and thus, we can draw an instance edge to H . This finishes the graph construction for `iterate`.

4.2 Proving Termination of Algorithms with Definite Reachability

The method `iterate` terminates since the sublist between `x` and `this` is shortened in every loop iteration. To extract this argument automatically, we proceed similar to Sect. B, i.e., we extend the states by suitable *counters*. More precisely, any state that contains a definite reachability annotation $o \xrightarrow{FI}! o'$ is extended by a counter $c_{o \xrightarrow{FI}! o'}$ representing the length of the FI -path from o to o' .

¹⁰ In the graph, we have shortened `this` to `t`.

¹¹ This annotation roughly corresponds to $ls(o_2, o_1)$ in separation logic, cf. e.g. [45].

So H is extended by two counters $c_{o_1 \xrightarrow{\{n\}} o_4}$ and $c_{o_4 \xrightarrow{\{n\}} o_1}$. Information about their value can only be inferred when we perform a *refinement* or when we *transfer* an annotation $o \xrightarrow{FI} o'$ to some successor \hat{o} of o' (yielding $o \xrightarrow{FI} \hat{o}$).

If a state s contains both $o \xrightarrow{FI} o'$ and $o =? o'$, then an *equality refinement* according to $o =? o'$ yields two successor states. In one of them, o and o' are identified and $o \xrightarrow{FI} o'$ is removed. In the other successor state s' (for $o \neq o'$), any path from o to o' must have at least length one. Hence, if “ $c_{o \xrightarrow{FI} o'} : i$ ” in s and s' , then the edge from s to s' can be labeled by “ $i > 0$ ”. So in our example, if “ $c_{o_4 \xrightarrow{\{n\}} o_1} : i$ ” in I and J , then we can add “ $i > 0$ ” to the edge from I to J .

Moreover, if s contains $o \xrightarrow{FI} o'$ and one performs an *instance refinement* on o , then in each successor state s' of s , the annotation $o \xrightarrow{FI} o'$ is replaced by $\hat{o} \xrightarrow{FI} o'$ for the reference \hat{o} with $o.f = \hat{o}$ where $f \in FI$. Instead of “ $c_{o \xrightarrow{FI} o'} : i$ ” in s we now have a counter “ $c_{\hat{o} \xrightarrow{FI} o'} : i'$ ” in s' . Since FI is deterministic, the FI -path from \hat{o} to o' is one step shorter than the FI -path from o to o' . Thus, the edge from s to s' is labeled by “ $i' = i - 1$ ”. So if we have “ $c_{o_4 \xrightarrow{FI} o_1} : i$ ” in K and “ $c_{o_6 \xrightarrow{FI} o_1} : i$ ” in L , then we add “ $i' = i - 1$ ” to the edge from K to L .

When a reference o' has become unneeded in a state s' reached by evaluation from s , then we *transfer* annotations of the form $o \xrightarrow{FI} o'$ to all successors \hat{o} of o' with $o' \xrightarrow{FI'} \hat{o}$ where $FI' = \{f\} \cup FI$ is still deterministic. This results in a new annotation $o \xrightarrow{FI'} \hat{o}$ in s' . For “ $c_{o \xrightarrow{FI} o'} : i$ ” in s' , we know that its value is exactly one more than “ $c_{o \xrightarrow{FI} o} : i$ ” in s and hence, we label the edge by “ $i' = i + 1$ ”. In our example, this happens between L and H . Here the annotation $o_1 \xrightarrow{\{n\}} o_5$ is transferred to o_5 's successor o_6 when o_5 is garbage collected, yielding $o_1 \xrightarrow{\{n\}} o_6$. Thm. 4 adapts Thm. 1 to definite reachability annotations.

Theorem 4 (Soundness of Termination Graphs with Definite Reachability). Let G be a termination graph with definite reachability annotations, s_1 a state in G , and c_1 a concrete state with $c_1 \sqsubseteq s_1$. As in Thm. 1, any computation sequence c_1, c_2, \dots is represented by a path $s_1^1, \dots, s_1^{k_1}, s_2^1, \dots, s_2^{k_2}, \dots$ in G .

Let G' result from G by extending the states by counters for their definite reachability annotations as above. Moreover, each concrete state c_j in the computation sequence is extended to a concrete state c'_j by adding counters “ $c_{o \xrightarrow{FI} o'} : i$ ” for all annotations “ $o \xrightarrow{FI} o'$ ” in $s_j^1, \dots, s_j^{k_j}$. Here, the heap of c'_j maps i to the singleton interval containing the length of the FI -path between the references corresponding to o and o' in c'_j . Then the computation sequence c'_1, c'_2, \dots of these extended concrete states is represented by the termination graph G' .

The generation of TRSs from the termination graph works as before. Hence by Thm. 2 and 4, termination of the resulting TRSs implies that there is no infinite computation sequence c'_1, c'_2, \dots of extended concrete states and thus, also no infinite computation sequence c_1, c_2, \dots . Hence, the Java program is terminating. Moreover, Thm. 4 can also be combined with Thm. 3, i.e., the states may also contain counters for marking properties as in Thm. 3.

As in Sect. 3, the new counters result in extra arguments¹² of the function symbols in the TRS. In our example, we obtain the following TRS from the only SCC I, \dots, L, \dots, I (after “merging” some rules). Termination of this TRS is easy to prove automatically, which implies termination of `iterate`.

$$\begin{aligned} f_I(\dots, i, \dots) &\rightarrow f_K(\dots, i, \dots) \mid i > 0 & f_K(\dots, i, \dots) &\rightarrow f_L(\dots, i - 1, \dots) \\ f_L(\dots, i', \dots) &\rightarrow f_I(\dots, i', \dots) \end{aligned}$$

5 Experiments and Conclusion

We extended our earlier work [6, 7, 8, 25] on termination of Java to handle methods whose termination depends on cyclic data. We implemented our contributions in the tool AProVE [19] (using the SMT Solver Z3 [15]) and evaluated it on a collection of 387 JBC programs. It consists of all¹³ 268 Java programs of the *Termination Problem Data Base* (used in the *International Termination Competition*); the examples `length`, `visit`, `iterate` from this paper¹⁴ a variant of `visit` on graphs;¹⁵ 3 well-known challenge problems from [10]; 57 (non-terminating) examples from [8]; and all 60 methods of `java.util.LinkedList` and `java.util.HashMap` from Oracle’s standard Java distribution.¹⁶ Apart from list algorithms, the collection also contains many programs on integers, arrays, trees, or graphs. Below, we compare the new version of AProVE with AProVE ’11 (implementing [6, 7, 8, 25], i.e., without support for cyclic data), and with the other available termination tools for Java, viz. Julia [30] and COSTA [2]. As in the *Termination Competition*, we allowed a runtime of 60 seconds for each example. Since the tools are tuned to succeed quickly, the results hardly change when increasing the time-out. “Yes” resp. “No” states

how often termination was proved resp. disproved, “Fail” indicates failure in less than 60 seconds, “T” states how many examples led to a Time-out, and “R” gives the average Runtime in seconds for each example.

	Y	N	F	T	R
AProVE	267	81	11	28	9.5
AProVE ’11	225	81	45	36	11.4
Julia	191	22	174	0	4.7
COSTA	160	0	181	46	11.0

Our experiments show that AProVE is substantially more powerful than all other tools. In particular, AProVE succeeds for all problems of [10]¹⁷ and for 85 % of the examples from `LinkedList` and `HashMap`. There, AProVE ’11, Julia, resp. COSTA can only handle 38 %, 53 %, resp. 48 %. See [1] to access AProVE via a

¹² For reasons of space, we only depicted the argument for the counter $o_4 \xrightarrow{\{n\}} o_1$.

¹³ We removed one controversial example whose termination depends on overflows.

¹⁴ Our approach automatically infers with which input `length`, `visit`, and `iterate` are called, i.e., we automatically obtain the termination graphs in Fig. 4, 7, and 10.

¹⁵ Here, the technique of Sect. 3 succeeds and the one of Sect. 4 fails, cf. Footnote 6.

¹⁶ Following the regulations in the *Termination Competition*, we excluded 7 methods from `LinkedList` and `HashMap`, as they use native methods or string manipulation.

¹⁷ We are not aware of any other tool that proves termination of the algorithm for in-place reversal of pan-handle lists from [10] fully automatically.

web interface, for the examples and details on the experiments, and for [6,7,8,9,25].

Acknowledgements. We thank F. Spoto and S. Genaim for help with the experiments and A. Rybalchenko and the anonymous referees for helpful comments.

References

1. <http://aprove.informatik.rwth-aachen.de/eval/JBC-Cyclic/>
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)
3. Baader, F., Nipkow, T.: Term Rewriting and All That, Cambridge (1998)
4. Berdine, J., Cook, B., Distefano, D., O’Hearn, P.W.: Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
5. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O’Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
6. Brockschmidt, M., Otto, C., von Essen, C., Giesl, J.: Termination Graphs for Java Bytecode. In: Siegler, S., Wasser, N. (eds.) Walther Festschrift. LNCS, vol. 6463, pp. 17–37. Springer, Heidelberg (2010)
7. Brockschmidt, M., Otto, C., Giesl, J.: Modular termination proofs of recursive JBC programs by term rewriting. In: Proc. RTA 2011. LIPIcs, vol. 10, pp. 155–170 (2011)
8. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for JBC. In: Proc. FoVeOOS 2011. LNCS (2012)
9. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. Technical Report AIB 2012-06, RWTH Aachen (2012), Available from [1] and from <http://aib.informatik.rwth-aachen.de>
10. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proc. POPL 2008, pp. 101–112. ACM Press (2008)
11. Cherini, R., Rearte, L., Blanco, J.: A Shape Analysis for Non-linear Data Structures. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 201–217. Springer, Heidelberg (2010)
12. Colón, M.A., Sipma, H.B.: Practical Methods for Proving Program Termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
13. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. PLDI 2006, pp. 415–426. ACM Press (2006)
14. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL 1977, pp. 238–252. ACM Press (1977)
15. 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)
16. Dershowitz, N.: Termination of rewriting. J. Symb. Comp. 3(1-2), 69–116 (1987)
17. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Proc. RTA 2011. LIPIcs, vol. 10, pp. 41–50 (2011)

18. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving Termination of Integer Term Rewriting. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
19. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
20. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. *Journal of Automated Reasoning* 37(3), 155–203 (2006)
21. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. *ACM TOPLAS* 33(2) (2011)
22. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination Analysis with Compositional Transition Invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
23. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proc. POPL 2001, pp. 81–92. ACM Press (2001)
24. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic numeric abstractions for heap-manipulating programs. In: Proc. POPL 2010, pp. 81–92. ACM Press (2010)
25. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of JBC by term rewriting. In: Proc. RTA 2010. LIPIcs, vol. 6, pp. 259–276 (2010)
26. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
27. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS 2004, pp. 32–41 (2004)
28. Podelski, A., Rybalchenko, A., Wies, T.: Heap Assumptions on Demand. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 314–327. Springer, Heidelberg (2008)
29. Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A., Thiemann, R.: Automated termination analysis for logic programs with cut. *TPLP* 10(4-6), 365–381 (2010)
30. Spoto, F., Mesnard, F., Payet, É.: A termination analyser for Java Bytecode based on path-length. *ACM TOPLAS* 32(3) (2010)
31. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop Summarization and Termination Analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)
32. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O’Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
33. Zantema, H.: Termination. In: Terese (ed.) *Term Rewriting Systems*, pp. 181–259. Cambridge University Press (2003)

Proving Termination of Probabilistic Programs Using Patterns

Javier Esparza¹, Andreas Gaiser^{1,*}, and Stefan Kiefer^{2,**}

¹ Institut für Informatik, Technische Universität München, Germany

{esparza,gaiser}@model.in.tum.de

² Department of Computer Science, University of Oxford, United Kingdom

stefan.kiefer@cs.ox.ac.uk

Abstract. Proving programs terminating is a fundamental computer science challenge. Recent research has produced powerful tools that can check a wide range of programs for termination. The analog for probabilistic programs, namely termination with probability one (“almost-sure termination”), is an equally important property for randomized algorithms and probabilistic protocols. We suggest a novel algorithm for proving almost-sure termination of probabilistic programs. Our algorithm exploits the power of state-of-the-art model checkers and termination provers for nonprobabilistic programs: it calls such tools within a refinement loop and thereby iteratively constructs a “terminating pattern”, which is a set of terminating runs with probability one. We report on various case studies illustrating the effectiveness of our algorithm. As a further application, our algorithm can improve lower bounds on reachability probabilities.

1 Introduction

Proving program termination is a fundamental challenge of computer science. Termination is expressible in temporal logic, and so checkable in principle by LTL or CTL model-checkers. However, recent research has shown that special purpose tools, like Terminator and ARMC [18, 4], and techniques like *transition invariants*, can be dramatically more efficient [17, 20, 19].

The analog of termination for probabilistic programs is termination with probability one, or *almost sure termination*, abbreviated here to *a.s.-termination*. Since a.s.-termination is as important for randomized algorithms and probabilistic protocols as termination is for regular programs, the question arises whether the very strong advances in automatic termination proving termination can be exploited in the probabilistic case. However, it is not difficult to see that, without further restricting the question, the answer is negative. The reason is that termination is a purely topological property of the transition system associated

* Andreas Gaiser is supported by the DFG Graduiertenkolleg 1480 (PUMA).

** Stefan Kiefer is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

to the program, namely absence of cycles, but a.s.-termination is not. Consider for instance the program

```
k = 1; while (0 < k) { if coin(p) k++ else k-- }
```

where `coin(p)` yields 1 with probability $0 < p < 1$, and 0 with probability $(1 - p)$. The program has the same executions for all values of p (only their probabilities change), but it only terminates a.s. for $p \leq 1/2$. This shows that proving a.s.-termination requires arithmetic reasoning not offered by termination provers.

The situation changes if we restrict our attention to *weakly finite* probabilistic programs. Loosely speaking, a program is weakly finite if the set of states reachable from any initial state is finite. Notice that the state space may be infinite, because the set of initial states may be infinite. Weakly finite programs are a large class, which in particular contains *parameterized probabilistic programs*, i.e., programs with parameters that can be initialized to arbitrary large values, but are finite-state for every valuation of the parameters. One can show that a.s.-termination is a topological property for weakly finite programs. If the program does not contain nondeterministic choices, then it terminates a.s. iff for every reachable state s there is a path leading from s to a terminating state, which corresponds to the CTL property *AG EF end*. (In the nondeterministic case there is also a corresponding topological property.) As in the nonprobabilistic case, generic infinite-state model checkers perform poorly for these properties because of the quantifier alternation *AG EF*. In particular, CEGAR approaches usually fail, because, crudely speaking, they tend to unroll loops, which is essentially useless for proving termination.

In [1], Arons, Pnueli and Zuck present a different and very elegant approach that reduces *a.s.-termination* of a *probabilistic* program to *termination* of a *nondeterministic* program obtained with the help of a *Planner*. A Planner occasionally and infinitely often determines the outcome of the next k random choices for some fixed k , while the other random choices are performed nondeterministically. The planner approach is based on the following simple proof rule, with P a probabilistic program and R a measurable set of runs of P :

$$\frac{Pr[R] = 1 \quad \text{Every } r \in R \text{ is terminating}}{P \text{ terminates a.s.}}$$

In this paper we revisit and generalize this approach, with the goal of profiting from recent advances on termination tools and techniques not available when [1] was published. While we also partially fix the outcome of random choices, we do so more flexibly with the help of *patterns*. A first advantage of patterns is that we are able to obtain a *completeness* result for weakly finite programs, which is not the case for Planners. Further, in contrast to [1], we show how to automatically derive patterns for finite-state and weakly finite programs using an adapted version of the CEGAR approach. Finally, we apply our technique to improve CEGAR-algorithms for *quantitative* probabilistic verification [7,8,10,5].

In the rest of this introduction we explain our approach by means of examples. First we discuss finite-state programs and then the weakly finite case.

Finite-state programs. Consider the finite-state program FW shown on the left of Fig. □. It is an abstraction of part of the FireWire protocol [12]. Loosely speaking,

```

c1 = ?; c2 = 2;
k = 0;
while (k < 100) {
    old_x = x;
    if (c1 > 0) { x = nondet(); c1-- }
    elseif (c2 = 2 ) { x = 0; c2-- }
    elseif (c2 = 1 ) { x = 1; c2-- }
    else /* c1 = 0 and c2 = 0 */ { c1 = ?; c2 = 2 }
    if (x != old_x) k++
}

```

Fig. 1. The programs FW and FW'

FW terminates a.s. because if we keep tossing a coin then with probability 1 we observe 100 times two consecutive tosses with the opposite outcome (we even see 100 times the outcome 01). More formally, let $C = \{0, 1\}$, and let us identify a *run* of FW (i.e., a terminating or infinite execution) with the sequence of 0's and 1's corresponding to the results of the coin tosses carried out during it. For instance, $(01)^{51}$ and $(001100)^{50}$ are terminating runs of FW, and 0^ω is a nonterminating run. FW terminates because the runs that are prefixes of $(C^*01)^\omega$ have probability 1, and all of them terminate. But it is easy to see that these are also the runs of the nondeterministic program FW' on the right of Fig. □ where $c = ?$ nondeterministically sets c to an arbitrary nonnegative integer. Since termination of FW' can easily be proved with the help of ARMC, we have proved a.s.-termination of FW.

We present an automatic procedure leading from FW to FW' based on the notion of *patterns*. A pattern is a subset of C^ω of the form $C^*w_1C^*w_2C^*w_3\dots$, where $w_1, w_2, \dots \in C^*$. We call a pattern *simple* if it is of the form $(C^*w)^\omega$. A pattern Φ is *terminating* (for a probabilistic program P) if all runs of P that *conform* to Φ , i.e., that are prefixes of words of Φ , terminate. In the paper we prove the following theorems:

- (1) For every pattern Φ and program P , the Φ -conforming runs of P have probability 1.
- (2) Every finite-state program has a simple terminating pattern.

By these results, we can show that FW terminates a.s. by finding a simple terminating pattern Φ , taking for P' a nondeterministic program whose runs are the Φ -conforming runs of P , and proving that P' terminates. In the paper we show how to automatically find Φ with the help of a finite-state model-checker (in our experiments we use SPIN). We sketch the procedure using FW as example. First we check if some run of FW conforms to $\Phi_0 = C^\omega$, i.e., if some run of FW is infinite, and get $v_1 = 0^\omega$ as answer. Using an algorithm provided in the paper, we compute a *spoiler* w_1 of v_1 : a finite word that is not an infix of v_1 . The algorithm

yields $w_1 = 1$. We now check if some run of FW conforms to $\Phi_1 = (C^*w_1)^\omega$, and get $v_2 = 1^\omega$ as counterexample, and construct a spoiler w_2 of both v_1 and v_2 : a finite word that is an infix of neither v_1^ω nor v_2^ω . We get $w_2 = 01$, and check if some run of FW conforms to $\Phi_2 = (C^*w_2)^\omega$. The checker finds no counterexamples, and so Φ_2 is terminating. In the paper we prove that the procedure is complete, i.e., produces a terminating pattern for any finite-state program that terminates a.s.

Weakly finite programs. We now address the main goal of the paper: proving a.s.-termination for weakly finite programs. Unfortunately, Proposition (2) no longer holds. Consider the random-walk program RW on the left of Fig. 2, where N is an input variable. RW terminates a.s., but we can easily show (by setting N

```

K = 2; c1 = ?; c2 = K;
k = 1;
while (0 < k < N) {
    if (c1 > 0) {
        if nondet() k++ else k--; c1--
    }
    elseif (c2 > 0) { k--; c2-- }
    else { K++; c1 = ?; c2 = K }
}

```

Fig. 2. The programs RW and RW'

to a large enough value) that no simple pattern is terminating. However, there is a terminating pattern, namely $\Phi = C^*00C^*000C^*0000\dots$: every Φ -conforming run terminates, whatever value N is set to. Since, by result (1), the Φ -conforming runs have probability 1 (intuitively, when tossing a coin we will eventually see longer and longer chains of 0's), RW terminates a.s. In the paper we show that this is not a coincidence by proving the following completeness result:

- (3) Every weakly finite program has a (not necessarily simple) terminating pattern.

In fact, we even prove the existence of a *universal* terminating pattern, i.e., a single pattern Φ_u such that for all weakly finite, a.s.-terminating probabilistic programs all Φ_u -conforming runs terminate. This gives a universal reduction of a.s.-termination to termination, but one that is not very useful in practice. In particular, since the universal pattern is universal, it is not tailored towards making the proof of any particular program simple. For this reason we propose a technique that reuses the procedure for finite-state programs, and extends it with an extrapolation step in order to produce a candidate for a terminating pattern. We sketch the procedure using RW as example. Let RW_i be the program RW with $N = i$. Since every RW_i is finite-state, we can find terminating patterns $\Phi_i = (C^*u_i)^\omega$ for a finite set of values of i , say for $i = 1, 2, 3, 4, 5$. We obtain $u_1 = u_2 = \epsilon$, $u_3 = 00$, $u_4 = 000$, $u_5 = 000$. We prove in the paper that Φ_i is

not only terminating for RW_i , but also for every RW_j with $j \leq i$. This suggests to extrapolate and take the pattern $\Phi = C^*00C^*000C^*0000\dots$ as a candidate for a terminating pattern for RW . We automatically construct the nondeterministic program RW' on the right of Fig. 2. Again, ARMC proves that RW' terminates, and so that RW terminates a.s.

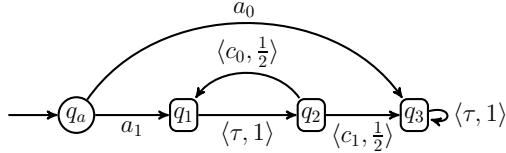
Related work. A.s.-termination is highly desirable for protocols if termination within a fixed number of steps is not feasible. For instance, [3] considers the problem of reaching consensus within a set of interconnected processes, some of which may be faulty or even malicious. They succeed in designing a probabilistic protocol to reach consensus a.s., although it is known that no deterministic algorithm terminates within a bounded number of steps. A well-known approach for proving a.s.-termination are Pnueli et al.'s notions of extreme fairness and α -fairness [15][16]. These proof methods, although complete for finite-state systems, are hard to automatize and require a lot of knowledge about the considered program. The same applies for the approach of McIver et al. in [11] that offers proof rules for probabilistic loops in pGCL, an extension of Dijkstra's guarded language. The paper [13] discusses probabilistic termination in an abstraction-interpretation framework. It focuses on programs with a (single) loop and proposes a method of proving that the probability of taking the loop k times decreases exponentially with k . This implies a.s.-termination. In contrast to our work there is no tool support in [13].

Organization of the paper. Sections 2 contains preliminaries and the syntax and semantics of our model of probabilistic programs. Section 3 proves soundness and completeness results for termination of weakly finite programs. Section 4 describes the iterative algorithm for generating patterns. Section 5 discusses case studies. Section 6 concludes. For space reasons, a full discussion of nondeterministic programs and some proofs are omitted. They can be found in the full version of the paper in [6].

2 Preliminaries

For a finite nonempty set Σ , we denote by Σ^* and Σ^ω the sets of finite and infinite words over Σ , and set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.

Markov Decision Processes and Markov Chains. A *Markov Decision Process* (MDP) is a tuple $\mathcal{M} = (Q_A, Q_P, \text{Init}, \rightarrow, \text{Lab}_A, \text{Lab}_P)$, where Q_A and Q_P are countable or finite sets of *action nodes* and *probabilistic nodes*, $\text{Init} \subseteq Q_A \cup Q_P$ is a set of *initial nodes*, and Lab_A and Lab_P are disjoint, finite sets of *action labels* and *probabilistic labels*. Finally, the relation \rightarrow is equal to $\rightarrow_A \cup \rightarrow_P$, where $\rightarrow_A \subseteq Q_A \times \text{Lab}_A \times (Q_A \cup Q_P)$ is a set of *action transitions*, and $\rightarrow_P \subseteq Q_P \times (0, 1] \times \text{Lab}_P \times Q$ is a set of *probabilistic transitions* satisfying the following conditions: (a) if (q, p, l, q') and (q, p', l, q') are probabilistic transitions, then $p = p'$; (b) the probabilities of the outgoing transitions of a probabilistic node add up to 1. We also require that every node of Q_A has at least one successor in \rightarrow_A . If $Q_A = \emptyset$ and $\text{Init} = \{q_I\}$ then we call \mathcal{M} a *Markov chain* and

**Fig. 3.** Example MDP

write $\mathcal{M} = (Q_P, q_I, \rightarrow, \text{Lab}_P)$. We set $Q = Q_A \cup Q_P$ and $\text{Lab} = \text{Lab}_A \cup \text{Lab}_P$. We write $q \xrightarrow{l} q'$ for $(q, l, q') \in \rightarrow_A$, and $q \xrightarrow{l,p} q'$ for $(q, p, l, q') \in \rightarrow_P$ (we skip p if it is irrelevant). For $w = l_1 l_2 \dots l_n \in \text{Lab}^*$, we write $q \xrightarrow{w} q'$ if there exists a path $q = q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_n} q_n = q'$.

Example 1. Figure 3 shows an example of a Markov Decision Process $\mathcal{M} = (\{q_a\}, \{q_1, q_2, q_3\}, \text{Init}, \rightarrow, \text{Lab}_A, \text{Lab}_P)$, with action labels a_0, a_1 , probabilistic labels τ, c_0, c_1 , and a single initial node q_a .

Runs, Paths, Probability Measures, Traces. A *run* of an MDP \mathcal{M} is an infinite word $r = q_0 l_0 q_1 l_1 \dots \in (Q\text{Lab})^\omega$ such that for all $i \geq 0$ either $q_i \xrightarrow{l_i, p} q_{i+1}$ for some $p \in (0, 1]$ or $q_i \xrightarrow{l_i} q_{i+1}$. We call the run *initial* if $q_0 \in \text{Init}$. We denote the set of runs starting at a node q by $\text{Runs}^\mathcal{M}(q)$, and the set of all runs starting at initial nodes by $\text{Runs}(\mathcal{M})$.

A *path* is a proper prefix of a run. We denote by $\text{Paths}^\mathcal{M}(q)$ the set of all paths starting at q . We often write $r = q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \dots$ instead of $r = q_0 l_0 q_1 \dots$ for both runs and paths, and skip the superscripts of $\text{Runs}(\cdot)$ and $\text{Paths}(\cdot)$ if the context is clear.

We take the usual, cylinder-based definition of a probability measure \Pr_{q_0} on the set of runs of a Markov chain \mathcal{M} starting at a state $q_0 \in \text{Init}$ (see e.g. [2] or [6] for details). For general MDPs, we define a probability measure $\Pr_{q_0}^S$ with respect to a *strategy* S . We may drop the subscript if the initial state is irrelevant or understood.

The *trace* of a run $r = q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \dots \in \text{Runs}(\mathcal{M})$, denoted by \bar{r} , is the infinite sequence $\alpha_0 \alpha_1 \dots \in \text{Lab}$ of labels. Given $\Sigma \subseteq \text{Lab}$, we define $\bar{r}|_\Sigma$ as the projection of \bar{r} onto Σ . Observe that $\bar{r}|_\Sigma$ can be finite.

2.1 Probabilistic Programs

We model probabilistic programs as flowgraphs whose transitions are labeled with *commands*. Since our model is standard and very similar to [10], we give an informal but hopefully precise enough definition. Let Var be a set of variable names over the integers (the variable domain could be easily extended), and let Val be the set of possible *valuations* of Var , also called *configurations*. The set of commands contains

- conditional statements, i.e., boolean combinations of expressions $e \leq e'$, where e, e' are arithmetic expressions (e.g., $x + y \leq 5 \wedge y \geq 3$);

- deterministic assignments $x := e$ and nondeterministic assignments $x := \text{nondet}()$ that nondeterministically assign to x the value 0 or 1;
- probabilistic assignments $x := \text{coin}(p)$ that assign to x the value 0 or 1 with probability p or $(1 - p)$, respectively.

A *probabilistic program* P is a tuple $(\mathcal{L}, I, \hookrightarrow, \text{label}, \perp, \top)$, where \mathcal{L} is a finite set of control flow *locations*, $I \subseteq \text{Val}$ is a set of *initial configurations*, $\hookrightarrow \subseteq \mathcal{L} \times \mathcal{L}$ is the *flow relation* (as usual we write $l \hookrightarrow l'$ for $(l, l') \in \hookrightarrow$, and call the elements of \hookrightarrow *edges*), label is a function that assigns a command to each edge, \perp is the *start location*, and \top is the *end location*. The following standard conditions must hold: (i) the only outgoing edge of \top is $\top \hookrightarrow \top$; (ii) either all or none of the outgoing edges of a location are labeled by conditional statements; if all, then every configuration satisfies the condition of exactly one outgoing edge; if none, then the location has exactly one outgoing edge; (iii) if an outgoing edge of a location is labeled by an assignment, then it is the only outgoing edge of this location. A location is *nondeterministic* if it has an outgoing edge labeled by a nondeterministic assignment, otherwise it is *deterministic*. Deterministic locations can be *probabilistic* or *nonprobabilistic*. A program is deterministic if all its locations are deterministic.

Program Semantics. The semantics of a probabilistic program is an MDP. Let P be a *probabilistic program* $(\mathcal{L}, I, \hookrightarrow, \text{label}, \perp, \top)$, and let $\mathcal{L}_D, \mathcal{L}_A$ denote the sets of deterministic and nondeterministic locations of P . The semantics of P is the MDP $\mathcal{M}_P := (Q_A, Q_D, \text{Init}, \rightarrow, \text{Lab}_A, \text{Lab}_P)$, where $Q_A = \mathcal{L}_A \times \text{Val}$ is the set of nondeterministic nodes, $Q_D = ((\mathcal{L} \setminus \mathcal{L}_A) \times \text{Val}) \cup \{\top\}$ is the set of deterministic nodes, $\text{Init} = \{\perp\} \times I$ is the set of initial nodes, $\text{Lab}_A = \{a_0, a_1\}$ is the set of action labels, $\text{Lab}_P = \{\tau, 0, 1\}$ is the set of probabilistic labels, and the relation \rightarrow is defined as follows: For every node $v = \langle l, \sigma \rangle$ of \mathcal{M}_P and every edge $l \hookrightarrow l'$ of P

- if $\text{label}(l, l') = (x := \text{coin}(p))$, then $v \xrightarrow{0,p} \langle l', \sigma[x \mapsto 0] \rangle$ and $v \xrightarrow{1,1-p} \langle l', \sigma[x \mapsto 1] \rangle$;
- if $\text{label}(l, l') = (x := \text{nondet}())$, then $v \xrightarrow{a_0} \langle l', \sigma[x \mapsto 0] \rangle$ and $v \xrightarrow{a_1} \langle l', \sigma[x \mapsto 1] \rangle$;
- if $\text{label}(l, l') = (x := e)$, then $v \xrightarrow{\tau,1} \langle l', \sigma[x \mapsto e(\sigma)] \rangle$, where $\sigma[x \mapsto e(\sigma)]$ denotes the configuration obtained from σ by updating the value of x to the expression e evaluated under σ ;
- if $\text{label}(l, l') = c$ for a conditional c satisfying σ , then $v \xrightarrow{\tau,1} \langle l', \sigma \rangle$.

For each node $v = \langle \top, \sigma \rangle$, $v \xrightarrow{\tau} \top$ and $\top \xrightarrow{\tau} \top$. □

A program $P = (\mathcal{L}, I, \hookrightarrow, \text{label}, \perp, \top)$ is called

- *a.s.-terminating* if $\Pr_q^S[\{r \in \text{Runs}(\mathcal{M}_P) \mid r \text{ reaches } \top\}] = 1$ for every strategy S and every initial state q of \mathcal{M}_P ;
- *finite* if finitely many nodes are reachable from the initial nodes of \mathcal{M}_P ;
- *weakly finite* if P_b is finite for all $b \in I$, where P_b is obtained from P by fixing b as the only initial node.

Assumption. We assume in the following that programs to be analyzed are deterministic. We consider nondeterministic programs only in Section 3.1.

3 Patterns

We introduce the notion of patterns for probabilistic programs. A pattern restricts a probabilistic program by imposing particular sequences of coin toss outcomes on the program runs. For the rest of the section we fix a probabilistic program $P = (\mathcal{L}, I, \hookrightarrow, \text{label}, \perp, \top)$ and its associated MDP $\mathcal{M}_P = (Q_A, Q_P, \text{Init}, \rightarrow, \text{Lab}_A, \text{Lab}_P)$.

We write $C := \{0, 1\}$ for the set of coin toss outcomes in the following. A *pattern* is a subset of C^ω of the form $C^*w_1C^*w_2C^*w_3\dots$, where $w_1, w_2, \dots \in \Sigma^*$. We say the sequence w_1, w_2, \dots induces the pattern. Fixing an enumeration x_1, x_2, \dots of C^* , we call the pattern induced by x_1, x_2, \dots the *universal* pattern. For a pattern Φ , a run $r \in \text{Runs}(\mathcal{M}_P)$ is Φ -conforming if there is $v \in \Phi$ such that $\bar{r}|_C$ is a prefix of v . We call a pattern Φ terminating (for P) if all Φ -conforming runs terminate, i.e., reach \top . We show the following theorem:

Theorem 2.

- (1) Let Φ be a pattern. The set of Φ -conforming runs has probability 1. In particular, if Φ is terminating, then P is a.s.-terminating.
- (2) If P is a.s.-terminating and weakly finite, then the universal pattern is terminating for P .
- (3) If P is a.s.-terminating and finite with $n < \infty$ reachable nodes in \mathcal{M}_P , then there exists a word $w \in C^*$ with $|w| \in \mathcal{O}(n^2)$ such that C^*wC^ω is terminating for P .

Part (1) of Theorem 2 is the basis for the pattern approach. It allows to ignore runs that are not Φ -conforming, because they have probability 0. Part (2) states that the pattern approach is “complete” for a.s.-termination and weakly finite programs: For any a.s.-terminating and weakly finite program there is a terminating pattern; moreover the universal pattern suffices. Part (3) refines part (2) for finite programs: there is a short word such that C^*wC^ω is terminating.

Proof (of Theorem 2).

Part (1) (Sketch): We can show that the set of runs r that visit infinitely many probabilistic nodes and do not have the form $C^*w_1C^\omega$ is a null set. This result can then easily be generalized to $C^*w_1C^*w_2\dots C^*w_nC^\omega$. All runs conforming Φ can then be formed as a countable intersection of such run sets.

Part (2): Let $\sigma_1, \sigma_2, \dots$ be a (countable or infinite) enumeration of the nodes in I . With Part (3) we obtain for each $i \geq 1$ a word w_i such that $C^*w_iC^\omega$ is a terminating pattern for P , if the only starting node considered is σ_i . By its definition, the universal pattern is a subset of $C^*w_iC^\omega$ for every $i \geq 1$, so it is also terminating.

Part (3) (Sketch): Since P is a.s.-terminating, for every node q there exists a coin toss sequence w_q , $|w_q| \leq n$, with the following property: a run that passes

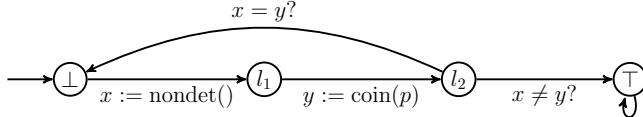


Fig. 4. Nondeterministic a.s.-terminating program without terminating pattern

through q and afterwards visits exactly the sequence w_q of coin toss outcomes is terminating. We build a sequence w such that for every state q every run that passes through q and then visits exactly the sequence w is terminating. We start with $w = w_q$ for an arbitrary $q \neq \top$. Then we pick a $q' \neq \top$ such that for $q'' \neq q$, runs starting in q'' and visiting exactly the probabilistic label sequence w lead to q' . We set $w = w_q w_{q'}$; after visiting w , all runs starting from q and q' end in \top . We iterate this until no more q' can be found. We stop after at most n steps and obtain a sequence w of length $\leq n^2$. \square

3.1 Nondeterministic Programs

For nondeterministic a.s.-terminating programs, there might not exist a terminating pattern, even if the program is finite. Figure 4 shows an example. Let Φ be a pattern and $c_1 c_2 c_3 \dots \in \Phi$. The run

$$\langle \perp, \sigma_0 \rangle \xrightarrow{a_{c_1}} \langle l_1, \sigma_1 \rangle \xrightarrow{c_1} \langle l_2, \sigma'_1 \rangle \xrightarrow{\tau} \langle \perp, \sigma'_1 \rangle \xrightarrow{a_{c_2}} \langle l_1, \sigma_2 \rangle \xrightarrow{c_2} \langle l_2, \sigma'_2 \rangle \xrightarrow{\tau} \langle \perp, \sigma'_2 \rangle \xrightarrow{a_{c_3}} \dots$$

in \mathcal{M}_P is Φ -conforming but nonterminating.

We show that the concept of patterns can be suitably generalized to nondeterministic programs, recovering a close analog of Theorem 2. Assume that the program is in a normal form where nondeterministic and probabilistic locations strictly alternate. This is easily achieved by adding dummy assignments. Writing $A := \{a_0, a_1\}$, every run $r \in \mathcal{M}_P$ satisfies $r|_{A \cup C} \in (AC)^\infty$.

A *response* of length n encodes a mapping $A^n \rightarrow C^n$ in an “interleaved” fashion, e.g., $\{a_01, a_10\}$ is a response of length one, $\{a_00a_01, a_00a_11, a_10a_01, a_10a_11\}$ is a response of length two. A *response pattern* is a subset of $(AC)^\omega$ of the form $(AC)^* R_1 (AC)^* R_2 (AC)^* \dots$, where R_1, R_2, \dots are responses. If we now define the notions of *universal* and *terminating* response patterns analogously to the deterministic case, a theorem very much like Theorem 2 can be shown. For instance, let $\Phi = (AC)^* \{a_01, a_10\} (AC)^\omega$. Then every Φ -conforming run of the program in Fig. 4 has the form

$$\langle \perp, \sigma_0 \rangle \rightarrow \dots \rightarrow q \xrightarrow{a_i} q' \xrightarrow{1-i} q'' \rightarrow \top \rightarrow \dots \quad \text{for an } i \in \{0, 1\}.$$

This implies that the program is a.s.-terminating. See [6] for the details.

4 Our Algorithm

In this section we aim at a procedure that, given a weakly finite program P , proves that P is a.s.-terminating by computing a terminating pattern. This

approach is justified by Theorem 2 (1). In fact, the proof of Theorem 2 (3) constructs, for any finite a.s.-terminating program, a terminating pattern. However, the construction operates on the Markov chain \mathcal{M}_P , which is expensive to compute. To avoid this, we would like to devise a procedure which operates on P , utilizing (nonprobabilistic) verification tools, such as model checkers and termination provers.

Theorem 2 (2) guarantees that, for any weakly finite a.s.-terminating program, the universal pattern is terminating. This suggests the following method for proving a.s.-termination of P : (i) replace in P all probabilistic assignments by nondeterministic ones and instrument the program so that all its runs are conforming to the universal pattern (this can be done as we describe in Section 4.1 below); then (ii) check the resulting program for termination with a termination checker such as ARMC [18]. Although this approach is sound and complete (modulo the strength of the termination checker), it turns out to be useless in practice. This is because the crucial loop invariants are extremely hard to catch for termination checkers. Already the instrumentation that produces the enumeration of C^* requires a nontrivial procedure (such as a binary counter) whose loops are difficult to analyze.

Therefore we devise in the following another algorithm which tries to compute a terminating pattern $C^*w_1C^*w_2\dots$. It operates on P and is “refinement”-based. Our algorithm uses a “pattern checker” subroutine which takes a sequence w_1, w_2, \dots , and checks (or attempts to check) whether the induced pattern is terminating. If it is not, the pattern checker may return a *lasso* as counterexample. Formally, a lasso is a sequence

$$\langle l_1, \sigma_1 \rangle \rightarrow \langle l_2, \sigma_2 \rangle \rightarrow \dots \rightarrow \langle l_m, \sigma_m \rangle \rightarrow \dots \rightarrow \langle l_n, \sigma_n \rangle \quad \text{with } \langle l_n, \sigma_n \rangle \rightarrow \langle l_m, \sigma_m \rangle$$

and $\langle l_1, \sigma_1 \rangle \in \text{Init}$. We call the sequence $\langle l_m, \sigma_m \rangle \rightarrow \dots \rightarrow \langle l_n, \sigma_n \rangle$ the *lasso loop* of the lasso. Note that a lasso naturally induces a run in $\text{Runs}(\mathcal{M}_P)$. If P is finite, pattern checkers can be made complete, i.e., they either prove the pattern terminating or return a lasso.

We present our pattern-finding algorithms for finite-state and weakly finite programs. In Section 4.1 we describe how pattern-finding and pattern-checking can be implemented using existing verification tools.

Finite Programs. First we assume that the given program P is finite. The algorithm may take a *base word* $s_0 \in C^*$ as input, which is set to $s_0 = \epsilon$ by default. Then it runs the pattern checker on $C^*s_0C^*s_0\dots$. If the pattern checker shows the pattern terminating, then, by Theorem 2 (1), P is a.s.-terminating. Otherwise the pattern checker provides a lasso $\langle l_1, \sigma_1 \rangle \rightarrow \dots \rightarrow \langle l_m, \sigma_m \rangle \rightarrow \dots \rightarrow \langle l_n, \sigma_n \rangle$. Our algorithm extracts from the lasso loop a word $u_1 \in C^*$, which indicates a sequence of outcomes of the coin tosses in the lasso loop. If $u_1 = \epsilon$, then the pattern checker has found a nonterminating run with only finitely many coin tosses, hence P is not a.s.-terminating. Otherwise (i.e., $u_1 \neq \epsilon$), let $s_1 \in C^*$ be a shortest word such that s_0 is a prefix of s_1 and s_1 is not an infix of u_1^ω . Our algorithm runs the pattern checker on $C^*s_1C^*s_1\dots$. If the pattern checker shows the pattern terminating, then P is a.s.-terminating. Otherwise

the pattern checker provides another lasso, from which our algorithm extracts a word $u_2 \in C^*$ similarly as before. If $u_2 = \epsilon$, then P is not a.s.-terminating. Otherwise, let $s_2 \in C^*$ be a shortest word such that s_0 is a prefix of s_2 and s_2 is neither an infix of u_1^ω nor an infix of u_2^ω . Observe that the word s_1 is an infix of u_2^ω by construction, hence $s_2 \neq s_1$. Our algorithm runs the pattern checker on $C^*s_2C^*s_2\dots$ and continues similarly. More precisely, in the $i - th$ iteration it chooses s_i as a shortest word such that s_i is a prefix of s_{i-1} and s_i is not an infix of any of the words $u_1^\omega, \dots, u_i^\omega$, thus eliminating all lassos so far discovered.

The algorithm is complete for finite and a.s.-terminating programs:

Proposition 3. *Let P be finite and a.s.-terminating. Then the algorithm finds a shortest word w such that the pattern $C^*wC^*w\dots$ is terminating, thus proving termination of P .*

In each iteration the algorithm picks a word s_j that destroys all previously discovered lasso loops. If the loops are small, then the word is short:

Proposition 4. *We have $|s_j| \leq |s_0| + 1 + \log_2(|u_1| + \dots + |u_j|)$.*

The proofs for both propositions can be found in [6].

Weakly Finite Programs. Let us now assume that P is a.s.-terminating and weakly finite. We modify our algorithm. Let b_1, b_2, \dots be an enumeration of the set I of initial nodes. Our algorithm first fixes b_1 as the only initial node. This leads to a finite program, so we can run the previously described algorithm, yielding a word w_1 such that $C^*w_1C^*w_1\dots$ is terminating for the initial node b_1 . Next our algorithm fixes b_2 as the only initial node, and runs the previously described algorithm taking w_1 as base word. As before, this establishes a terminating pattern $C^*w_2C^*w_2\dots$ By construction of w_2 , the word w_1 is a prefix of w_2 , so the pattern $C^*w_1C^*w_2C^*w_2\dots$ is terminating for the initial nodes $\{b_1, b_2\}$. Continuing in this way we obtain a sequence w_1, w_2, \dots such that $C^*w_1C^*w_2\dots$ is terminating. Our algorithm may not terminate, because it may keep computing w_1, w_2, \dots . However, we will illustrate that it is promising to compute the first few w_i and then *guess* an expression for general w_i . For instance if $w_1 = 0$ and $w_2 = 00$, then one may guess $w_i = 0^i$. We encode the guessed sequence w_1, w_2, \dots in a finite way and pass the obtained pattern $C^*w_1C^*w_2\dots$ to a pattern checker, which may show the pattern terminating, establishing a.s.-termination of the weakly finite program P .

4.1 Implementing Pattern Checkers

Finite Programs. We describe how to build a pattern checker for finite programs P and patterns of the form $C^*wC^*w\dots$ We employ a model checker for finite-state nonprobabilistic programs that can verify temporal properties: Given as input a finite program and a Büchi automaton \mathcal{A} , the model checker returns a lasso if there is a program run accepted by \mathcal{A} (such runs are called “counterexamples” in classical terminology). Otherwise it states that there is no counterexample. For our case studies, we use the SPIN tool [9].

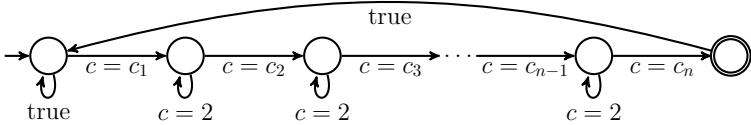


Fig. 5. Büchi automaton $\mathcal{A}(w)$, for $w = c_1 c_2 \dots c_n \in C^*$. Note that the number of states in $\mathcal{A}(w)$ grows linearly in $|w|$.

Given a finite probabilistic program P and a pattern $\Phi = C^* w C^* w \dots$, we first transform P into a nonprobabilistic program P' as follows. We introduce two fresh variables c and term , with ranges $\{0, 1, 2\}$ and $\{0, 1\}$, respectively, and add assignments $\text{term} = 0$ and $\text{term} = 1$ at the beginning and end of the program, respectively. Then every location l of P with $\text{label}(l, l') = x = \text{coin}(p)$ for a label l' is replaced by a nondeterministic choice and an if-statement as follows:

```
x = nondet();
if (x = 0) { c = 0; c = 2 } else { c = 1; c = 2 } end if;
```

In this way we can distinguish coin toss outcomes in a program trace by inspecting the assignments to c . Now we perform two checks on the nonprobabilistic program P' : First, we use SPIN to translate the LTL formula $G \neg \text{term} \wedge FG(c \notin \{0, 1\})$ into a Büchi automaton and check whether P' has a run that satisfies this formula. If there is indeed a lasso, our pattern checker reports it. Observe that by the construction of the LTL formula the lasso encodes a nonterminating run in P that eventually stops visiting probabilistic locations. So the lasso loop does not contain any coin tosses (and our algorithm will later correctly report that P is not a.s.-terminating). Otherwise, i.e., if no run satisfies the formula, we know that all nonterminating runs involve infinitely many coin tosses. Then we perform a second query: We construct a Büchi automaton $\mathcal{A}(w)$ that represents the set of infinite Φ -conforming runs, see Fig. 5. We use SPIN to check whether P' has run that is accepted by $\mathcal{A}(w)$. If yes, then there is an infinite Φ -conforming run, and our pattern checker reports the lasso. Otherwise, it reports that Φ is a terminating pattern.

Weakly Finite Programs. Recall that for weakly finite programs, the pattern checker needs to handle patterns of a more general form, namely $\Phi = C^* w_1 C^* w_2 \dots$. Even simple patterns like $C^* 0 C^* 00 C^* 000 \dots$ cannot be represented by a finite Büchi automaton. Therefore we need a more involved instrumentation of the program to restrict its runs to Φ -conforming ones. Now our pattern checker employs a termination checker for infinite-state programs. For our experiments we use ARMC.

Given a weakly finite program P and a pattern $\Phi = C^* w_1 C^* w_2 \dots$, we transform P into a nonprobabilistic program P^Φ as follows. We will use a command $x = ?$, which nondeterministically assigns a nonnegative integer to x . Further we assume that we can access the k -th letter of the i -th element of $(w_i)_{i \in \mathbb{N}}$ by $w[i][k]$, and $|w_i|$ by $\text{length}(w[i])$. We add fresh variables ctr , next and pos , where ctr is initialized nondeterministically with any nonnegative integer and

```

x = nondet();
if (ctr <= 0)
  if (pos > length(w[next])) { ctr = ?; pos = 1; next = next+1 }
  else { x = w[next][pos]; pos = pos+1 }
else ctr = ctr-1

```

Fig. 6. Code transformation for coin tosses in weakly finite programs

`next` and `pos` are both initialized with 1. If a run r is Φ -conforming, $\bar{r}|_C$ is a prefix of $v_1w_1v_2w_2v_3w_3\dots$, with $v_i \in C^*$. The variable `ctr` is used to “guess” the length of the words v_i ; the individual letters in v_i are irrelevant. We replace every command $c = \text{coin}(p)$ by the code sequence given in Fig. 6.

The runs in the resulting program P^Φ correspond exactly to the Φ -conforming runs in P . Then P^Φ is given to the termination checker. If it proves termination, we report “ Φ is a terminating pattern for P ”. Otherwise, the tool might either return a lasso, which our pattern checker reports, or give up on P^Φ , in which case our pattern checker also has to give up.

In our experiments, a weakly finite program typically has an uninitialized integer variable N whose value is nondeterministically fixed in the beginning. The pattern $C^*w_1C^*\dots C^*w_NC^\omega$ is then often terminating, which makes $\text{next} \leq N$ an invariant in P^Φ . The termination checker ARMC may benefit from this invariant, but may not be able to find it automatically (for reasons unknown to the authors). We therefore enhanced ARMC to “help itself” by adding the invariant $\text{next} \leq N$ to the program if ARMC’s reachability mode can verify the invariant.

5 Experimental Evaluation

We apply our methods to several parameterized programs taken from the literature.¹

- *firewire*: Fragment of FireWire’s symmetry-breaking protocol, adapted from [12] (a simpler version was used in the introduction). Roughly speaking, the number 100 of Fig. 11 is replaced by a parameter N .
- *randomwalk*: A slightly different version of the finite-range, one-dimensional random walk used as second example in the introduction.
- *herman*: An abstraction of Herman’s randomized algorithm for leader election used in [14]. It can be seen as a more complicated finite random walk, with N as the walk’s length.
- *zeroconf*: A model of the Zeroconf protocol taken from [10]. The protocol assigns IP addresses in a network. The parameter N is the number of probes sent after choosing an IP address to check whether it is already in use.
- *brp*: A model adapted from [10] that models the well-known bounded retransmission protocol. The original version can be proven a.s.-terminating with the trivial pattern C^ω ; hence we study an “unbounded” version, where arbitrarily many retransmissions are allowed. The parameter N is the length of the message that the sender must transmit to the receiver.

¹ The sources can be found at <http://www.model.in.tum.de/~gaiser/cav2012.html>

Name	#loc	Pattern words for $N = 1, 2, 3, 4$				Time (SPIN)	i -th word of guessed pattern	Time (ARMC)
<i>firewire</i>	19	010	010	010	010	17 sec	010	1 min 36 sec
<i>randomwalk</i>	16	ϵ	0^2	0^3	0^4	23 sec	0^i	1 min 22 sec
<i>herman</i>	36	010	$0(10)^2$	$0(10)^3$	$0(10)^4$	47 sec	$0(10)^i$	7 min 43 sec
<i>zeroconf</i>	39	0^3	0^4	0^5	0^6	20 sec	0^{i+2}	26 min 16 sec
<i>brp</i>	57	00	00	00	00	19 sec	00	45 min 14 sec

Fig. 7. Constructed patterns of the case studies and runtimes

Proving a.s.-termination. We prove a.s.-termination of the examples using SPIN [9] to find patterns of finite-state instances, and ARMC [18] to prove termination of the nondeterministic programs derived from the guessed pattern. All experiments were performed on an Intel[®] i7 machine with 8GB RAM. The results are shown in Fig. 7. The first two columns give the name of the example and its size. The next two columns show the words w_1, \dots, w_4 of the terminating patterns $C^*w_1C^\omega, \dots, C^*w_4C^\omega$ computed for $N = 1, 2, 3, 4$ (see Theorem 2(3) and Section 4.1), and SPIN’s runtime. The last two columns give word w_i in the guessed pattern $C^*w_1C^*w_2C^*w_3\dots$ (see Section 4.1), and ARMC’s runtime. For instance, the entry $0(10)^i$ for *herman* indicates that the guessed pattern is $C^*010C^*01010C^*0101010\dots$.

We derive two conclusions. First, a.s.-termination is proved by very simple patterns: the general shape is easily guessed from patterns for $N = 1, 2, 3, 4$, and the need for human ingenuity is virtually reduced to zero. This speaks in favor of the Planner technique of [1] and our extension to patterns, compared to other approaches using fairness and Hoare calculus [16, 11]. Second, the runtime is dominated by the termination tool, not by the finite-state checker. So the most direct way to improve the efficiency of our technique is to produce faster termination checkers.

In the introduction we claimed that general purpose probabilistic model-checkers perform poorly for a.s.-termination, since they are not geared towards this problem. To supply some evidence for this, we tried to prove a.s.-termination of the first four examples using the CEGAR-based PASS model checker [7, 8]. In all four cases the refinement loop did not terminate.²

Improving Lower Bounds for Reachability. Consider a program of the form `if coin(0.8) P1() else P2(); ERROR`. Probabilistic model-checkers compute lower and upper bounds for the probability of ERROR. Loosely speaking, lower bounds are computed by adding the probabilities of terminating runs of P1 and P2. However, since CEGAR-based checkers [7, 8, 10, 5] work with abstractions of P1 and P2, they may not be able to ascertain that paths of the abstraction are concrete paths of the program, leading to poor lower bounds. Information on a.s.-termination helps: if e.g. P1 terminates a.s., then we already have a lower

² Other checkers, like PRISM, cannot be applied because they only work for finite-state systems.

bound of 0.8. We demonstrate this technique on two examples. The first one is the following modification of *firewire*:

```
N = 1000; k = 0; miss = 0;
while (k < N) {
    old_x = x; x = coin(0.5);
    if (x = old_x) k++ else if (k < 5) miss = 1
}
```

For $i \in \{0, 1\}$, let p_i be the probability that the program terminates with $\text{miss} = i$. After 20 refinement steps PASS returns upper bounds of 0.032 for p_0 and 0.969 for p_1 , but a lower bound of 0 for p_1 , which stays 0 after 300 iterations. Our algorithm establishes that the loop a.s.-terminates, which implies $p_0 + p_1 = 1$, and so after 20 iterations we already get $0.968 \leq p_1 \leq 0.969$.

We apply the same technique to estimate the probabilities p_1, p_0 that *zeroconf* detects/does-not-detect an unused IP address. For $N = 100$, after 20 refinement steps PASS reports an upper bound of 0.999 for p_0 , but a lower bound of 0 for p_1 , which stays 0 for 80 more iterations. With our technique after 20 iterations we get $0.958 \leq p_1 \leq 0.999$.

6 Conclusions

We have presented an approach for automatically proving a.s.-termination of probabilistic programs. Inspired by the Planner approach of [1], we instrument a probabilistic program P into a nondeterministic program P' such that the runs of P' correspond to a set of runs of P with probability 1. The instrumentation is fully automatic for finite-state programs, and requires an extrapolation step for weakly finite programs. We automatically check termination of P' profiting from new tools that were not available to [1]. While our approach maintains the intuitive appeal of the Planner approach, it allows to prove completeness results. Furthermore, while in [1] the design of the Planner was left to the verifier, we have provided in our paper a CEGAR-like approach. In the case of parameterized programs, the approach requires an extrapolation step, which however in our case studies proved to be straightforward. Finally, we have also shown that our approach to improve the game-based CEGAR technique of [7,8,10] for computing upper and lower bounds for the probability of reaching a program location. While this technique often provides good upper bounds, the lower bounds are not so satisfactory (often 0), due to spurious nonterminating runs introduced by the abstraction. Our approach allows to remove the effect of these runs.

In future work we plan to apply learning techniques to pattern generation, thereby inferring probabilistic termination arguments for large program instances from small instances.

Acknowledgments. We thank the referees for helping us clarify certain aspects of the paper, Cornelius Popescu and Andrey Rybalchenko for many discussions and their help with ARMC, and Björn Wachter and Florian Zuleger for fruitful insights on quantitative probabilistic analysis and termination techniques.

References

1. Arons, T., Pnueli, A., Zuck, L.D.: Parameterized Verification by Probabilistic Abstraction. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 87–102. Springer, Heidelberg (2003)
2. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)
3. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32, 824–840 (1985)
4. Cook, B., Podelski, A., Rybalchenko, A.: TERMINATOR: Beyond Safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)
5. Esparza, J., Gaiser, A.: Probabilistic Abstractions with Arbitrary Domains. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 334–350. Springer, Heidelberg (2011)
6. Esparza, J., Gaiser, A., Kiefer, S.: Proving termination of probabilistic programs using patterns. Technical report (2012), <http://arxiv.org/abs/1204.2932>
7. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: Abstraction Refinement for Infinite Probabilistic Models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010)
8. 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)
9. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional (2003)
10. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: Abstraction Refinement for Probabilistic Software. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 182–197. Springer, Heidelberg (2009)
11. McIver, A., Morgan, C.: Developing and Reasoning About Probabilistic Programs in *pGCL*. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 123–155. Springer, Heidelberg (2006)
12. McIver, A., Morgan, C., Hoang, T.S.: Probabilistic Termination in *B*. In: Bert, D., Bowen, J. P., King, S., Waldén, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 216–239. Springer, Heidelberg (2003)
13. Monniaux, D.: An Abstract Analysis of the Probabilistic Termination of Programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 111–126. Springer, Heidelberg (2001)
14. Nakata, T.: On the expected time for Herman’s probabilistic self-stabilizing algorithm. Theoretical Computer Science 349(3), 475–483 (2005)
15. Pnueli, A.: On the extremely fair treatment of probabilistic algorithms. In: STOC, pp. 278–290. ACM (1983)
16. Pnueli, A., Zuck, L.D.: Probabilistic verification. Inf. Comput. 103, 1–29 (1993)
17. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society (2004)
18. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2006)
19. Podelski, A., Rybalchenko, A.: Transition Invariants and Transition Predicate Abstraction for Program Termination. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 3–10. Springer, Heidelberg (2011)
20. Rybalchenko, A.: Temporal verification with transition invariants. PhD thesis (2005)

The Gauge Domain: Scalable Analysis of Linear Inequality Invariants

Arnaud J. Venet

Carnegie Mellon University NASA Ames Research Center
Moffett Field, CA 94035
`arnaud.j.venet@nasa.gov`

Abstract. The inference of linear inequality invariants among variables of a program plays an important role in static analysis. The polyhedral abstract domain introduced by Cousot and Halbwachs in 1978 provides an elegant and precise solution to this problem. However, the computational complexity of higher-dimensional convex hull algorithms makes it impractical for real-size programs. In the past decade, much attention has been devoted to finding efficient alternatives by trading expressiveness for performance. However, polynomial-time algorithms are still too costly to use for large-scale programs, whereas the full expressive power of general linear inequalities is required in many practical cases. In this paper, we introduce the gauge domain, which enables the efficient inference of general linear inequality invariants within loops. The idea behind this domain consists of breaking down an invariant into a set of linear relations between each program variable and all loop counters in scope. Using this abstraction, the complexity of domain operations is no larger than $O(kn)$, where n is the number of variables and k is the maximum depth of loop nests. We demonstrate the effectiveness of this domain on a real 144K LOC intelligent flight control system, which implements advanced adaptive avionics.

1 Introduction

The discovery of numerical relationships among integer variables within a loop is one of the most fundamental tasks in formal software verification. Without this piece of information it would be impossible, for example, to analyze pointer arithmetic as it appears in real C programs. A fully automated solution based on convex polyhedra has been proposed by Cousot and Halbwachs [11] in what probably remains the most spectacular application of Abstract Interpretation. The polyhedral abstraction is precise enough to infer the exact invariants for most program loops in practice. It is based on the double description method [4, 21], which requires enumerating all faces of a convex polyhedron in all dimensions, an operation that has exponential time complexity in the worst case. Unfortunately, the combinatorial explosion almost always occurs in practice and this analysis cannot be reasonably applied to codes involving more than 15 or so variables.

Attempts have been made to improve the performance of the polyhedral domain. They essentially consist in finding more tractable albeit less precise

<pre> p = &msg; for (i = 0; i < n; i++) { if(*p == ...) { ... p += 16; } else { ... p += 32; } } </pre>	Convex polyhedra: $\begin{cases} 0 \leq i \leq n - 1 \\ 16i \leq p \leq 32i \end{cases}$	Gauges: $\begin{cases} \lambda \leq i \leq \lambda \\ 16\lambda \leq p \leq 32\lambda \end{cases}$
		Additional properties: $\begin{cases} \lambda \leq n - 1 \\ \lambda \in [0, +\infty] \end{cases}$

Fig. 1. Loop invariant expressed with convex polyhedra and gauges

alternatives to those domain operations that may exhibit exponential complexity (join, projection) without modifying the expressiveness of the domain itself [22, 26]. Linear programming techniques are used instead of the double-description method to compute approximate versions of operations on polyhedra. The idea is that the Simplex algorithm exhibits better runtime performance in practice, although still exponential in the worst case. However, available experimental data make it difficult to predict how these techniques would scale to real applications.

Another and more popular approach consists in identifying a subclass of convex polyhedra that possess better algorithmic properties. Notable domains include template polyhedra [24], octahedra [5], subpolyhedra [15], simplices [25], symbolic ranges [23] and the family of two-variables per inequality domains [17–20, 27]. Two members of the latter class, difference-bound matrices [18] and octagons [19], are particularly important since, to the best of our knowledge, they are the only general-purpose relational abstract domains that have been applied to the verification of large applications [1, 3, 10, 28].

Among relational domains that can express inequalities, octagons and difference-bound matrices have the lowest computational complexity: quadratic in space and cubic in time in the worst case. However, due to the nature of the closure algorithm employed to normalize their representation, the worst-case complexity is always attained in practice, which makes this kind of domain unusable for codes with more than a few dozen variables [28]. In order to address this issue, it is necessary to break down the set of program variables into small groups on which the abstract domain can be applied independently. This variable packing can be performed statically before analysis using knowledge on the application [10], or at analysis time, for example, by using dependency information computed on the fly [28].

However, the limited expressiveness of weakly relational domains precludes the direct analysis of pointer arithmetic, which requires more general forms of inequality constraints. This issue is addressed in C Global Surveyor [28] by using templates for access paths in data structures. The parameters appearing in the

template make up for the lack of expressive power of difference-bound matrices. Although effective, these techniques substantially complicate the construction of a static analyzer and they are very dependent on the characteristics of the code analyzed.

In our experience with analyzing large NASA codes, we have observed that most of the time, the value of a scalar variable inside a loop nest was entirely determined by the control structure in terms of symbolic bounds of the form $a_0 + a_1\lambda_1 + \dots + a_k\lambda_k$, where $\lambda_1, \dots, \lambda_k$ denote loop counters and a_1, \dots, a_k are integer coefficients. In this paper, we present an abstract interpretation framework in which each variable is approximated by a pair of such symbolic bounds, which we call a *gauge*. This abstraction generates far fewer constraints than weakly relational domains while providing greater expressiveness.

In Fig. 1 we have shown a code snippet that reads variable-sized data from a buffer of bytes, a common pattern in embedded programs. Gauges represent the implicit loop invariants, which are hard to infer, but do not say anything about loop bounds. The abstraction shall therefore be complemented with additional abstractions, like intervals and symbolic constants. The main idea is that it is far more efficient to combine simpler abstractions rather than have a powerful but inefficient domain take care of all properties at once. The gauge domain is not intended as a replacement for convex polyhedra or weakly relational domains, as it has limitations. However, it provides a simple and efficient way of generating precise loop invariants for a large swath of code without the need for customizing the static analyzer.

The paper is organized as follows. In Sect. 2 we formally define the gauge abstraction and state some of its basic properties. Section 3 introduces the Abstract Interpretation framework in which our analysis is specified. In Sect. 4 we construct an abstract domain that can infer gauge invariants on programs. Section 5 reports experimental results on a large NASA flight system. Section 6 concludes the paper.

2 The Gauge Abstraction

We now give a formal construction of gauges and characterize their natural ordering. Let $\Lambda = \{\lambda_1, \dots, \lambda_n\}$ be a fixed set of positive counters. Given integer coefficients a_0, \dots, a_n , we call *gauge bound* the expression $a_0 + \sum_{i=1}^n a_i\lambda_i$. Given a gauge bound g , we define the upper gauge \bar{g} as

$$\bar{g} = \{(x, l_1, \dots, l_n) \in \mathbb{Z} \times (\mathbb{Z}^+)^n \mid x \leq a_0 + \sum_{i=1}^n a_il_i\}$$

We define the *lower gauge* \underline{g} dually. Now, given two gauge bounds $g = a_0 + \sum_{i=1}^n a_i\lambda_i$ and $g' = a'_0 + \sum_{i=1}^n a'_i\lambda_i$, we would like to characterize the inclusion of upper gauges $\bar{g} \subseteq \bar{g}'$. This is equivalent to say that the following system has no integral solution:

$$S : \begin{cases} \lambda_i \geq 0 & i \in \{1, \dots, n\} \\ x \leq a_0 + a_1 \lambda_1 + \dots + a_n \lambda_n \\ x \geq 1 + a'_0 + a'_1 \lambda_1 + \dots + a'_n \lambda_n \end{cases}$$

First, observe that if $a'_0 < a_0$ then S has a trivial solution $\langle x = a'_0 + 1, \lambda_1 = 0, \dots, \lambda_n = 0 \rangle$. Assume for now that $a'_0 \geq a_0$ and S admits a rational solution $\langle x = u, \lambda_1 = l_1, \dots, \lambda_n = l_n \rangle$ such that all l_i are positive. There exists a nonzero positive integer μ such that $\mu u, \mu l_1, \dots, \mu l_n$ are all integers (take the lowest common multiplier of all denominators for example). We deduce from S the following inequalities:

$$\begin{cases} \mu u \leq \mu a_0 + a_1(\mu l_1) + \dots + a_n(\mu l_n) \\ \mu u \geq \mu(1 + a'_0) + a'_1(\mu l_1) + \dots + a'_n(\mu l_n) \end{cases}$$

which can be rewritten as:

$$\begin{cases} \mu u - (\mu - 1)a_0 \leq a_0 + a_1(\mu l_1) + \dots + a_n(\mu l_n) \\ \mu u - (\mu + (\mu - 1)a'_0) \geq a'_0 + a'_1(\mu l_1) + \dots + a'_n(\mu l_n) \end{cases}$$

From $a'_0 \geq a_0$ and $\mu \geq 1$ we deduce that

$$\mu + (\mu - 1)a'_0 > (\mu - 1)a_0$$

and then $\mu u - (\mu - 1)a_0 > \mu u - (\mu + (\mu - 1)a'_0)$. Therefore, the variable assignment

$$\langle x \mapsto \mu u - (\mu - 1)a_0, \lambda_1 \mapsto \mu l_1, \dots, \lambda_n \mapsto \mu l_n \rangle$$

is a solution of S . We just proved that if S admits a rational solution, then it also admits an integral solution. Therefore, S has no integral solution if and only if it has no rational solution. We can now reason entirely over rationals, which allows us to use a fundamental result of convex geometry, the Farkas lemma [29]:

Theorem 1 (Farkas Lemma). *Let $A \in \mathbb{Q}^{m \times d}$ and a column vector $\mathbf{z} \in \mathbb{Q}^m$. Either there exists a point $\mathbf{x} \in \mathbb{Q}^d$ with $A\mathbf{x} \leq \mathbf{z}$, or there exists a non-null row vector $\mathbf{c} \in (\mathbb{Q}^+)^m$, such that $\mathbf{c}A = \mathbf{0}$ and $\mathbf{c}\mathbf{z} < \mathbf{0}$.*

Note that, although originally established for real numbers, the Farkas lemma can be proven using only elementary linear algebra [12] and therefore holds on rationals. We define the matrix $A \in \mathbb{Q}^{(n+2) \times (n+1)}$ as follows:

$$A = \begin{pmatrix} -1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & -1 & 0 \\ -a_1 & \cdots & -a_{n-1} & -a_n & 1 \\ a'_1 & \cdots & a'_{n-1} & a'_n & -1 \end{pmatrix}$$

Let \mathbf{x} be the $(n + 1)$ -column vector and \mathbf{z} the $(n + 2)$ -column vector defined as:

$$\mathbf{x} = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_n \\ x \end{pmatrix} \text{ and } \mathbf{z} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a_0 \\ -a'_0 - 1 \end{pmatrix}$$

Then, the system S can be equivalently rewritten as:

$$A\mathbf{x} \leq \mathbf{z}$$

According to the Farkas lemma, S has no rational solution if and only if there exists a non-null $(n + 2)$ -row vector $\mathbf{c} = (c_1 \dots c_{n+2})$ of positive rationals such that $\mathbf{c}A = \mathbf{0}$ and $\mathbf{c}\mathbf{z} < \mathbf{0}$. If we unfold the matrix expression, this is equivalent to:

$$\begin{cases} -c_1 - c_{n+1}a_1 + c_{n+2}a'_1 = 0 \\ \vdots \quad \vdots \quad \vdots \\ -c_n - c_{n+1}a_n + c_{n+2}a'_n = 0 \\ c_{n+1} - c_{n+2} = 0 \\ c_{n+1}a_0 - c_{n+2}(a'_0 + 1) < 0 \end{cases}$$

If $c_{n+1} = 0$, then all c_i 's are equal to zero, which contradicts the fact that \mathbf{c} is non-null. Hence $c_{n+1} \neq 0$. Since c_1, \dots, c_n each appear in exactly one equation, we can recast this condition in a much simpler form. The system $A\mathbf{x} \leq \mathbf{z}$ has no rational solution if and only if there exists a rational number $c > 0$ such that

$$\begin{cases} c(a'_1 - a_1) \geq 0 \\ \vdots \quad \vdots \quad \vdots \\ c(a'_n - a_n) \geq 0 \\ c(a_0 - (a'_0 + 1)) < 0 \end{cases}$$

Since $c > 0$, this is equivalent to the following condition

$$\forall i \in \{0, \dots, n\} \quad a_i \leq a'_i$$

We can establish a similar result on lower gauges by duality.

Theorem 2. *If $g = a_0 + \sum_{i=1}^n a_i \lambda_i$ and $g' = a'_0 + \sum_{i=1}^n a'_i \lambda_i$, then $\underline{g} \subseteq \overline{g}'$ (resp. $\underline{g} \subseteq \underline{g}'$) iff $\forall i \in \{0, \dots, n\} \quad a_i \leq a'_i$ (resp. $a_i \geq a'_i$).*

By analogy with intervals, we define a gauge as a pair $[g, g']$ of gauge bounds and its denotation as $\underline{g} \cap \overline{g}'$. Note that a gauge is not empty if and only if, for all positive values of $\lambda_1, \dots, \lambda_n$, there is an $x \in \mathbb{Z}$ such that

$$a_0 + a_1 \lambda_1 + \dots + a_n \lambda_n \leq x \leq a'_0 + a'_1 \lambda_1 + \dots + a'_n \lambda_n$$

This condition can be equivalently restated as

$$a'_0 - a_0 + (a'_1 - a_1)\lambda_1 + \dots + (a'_n - a_n)\lambda_n \geq 0$$

for all positive values of $\lambda_1, \dots, \lambda_n$. An elementary reasoning shows that this property holds if and only if $a_i \leq a'_i$ for all $i \in \{0, \dots, n\}$, which is the exact analogue of the non-emptiness condition for intervals. We denote by $\perp_{\mathbf{G}}$ the empty gauge.

Now, given two non-empty gauges $G = [g_l, g_u]$ and $G' = [g'_l, g'_u]$, we need to characterize the inclusion of their denotation. Assume that $G \subseteq G'$. If $g_u = a_0 + \sum_{i=1}^n a_i \lambda_i$, then for all $(l_1, \dots, l_n) \in (\mathbb{Z}^+)^n$, we have $(a_0 + \sum_{i=1}^n a_i l_i, l_1, \dots, l_n) \in G$, because G is not empty. Since $G \subseteq G'$, we have $(a_0 + \sum_{i=1}^n a_i l_i, l_1, \dots, l_n) \in \overline{g}'_u$. By definition of upper gauges, this entails $\overline{g}_u \subseteq \overline{g}'_u$. By duality, we also have $\underline{g}_l \subseteq \underline{g}'_l$. We just proved the following result:

Theorem 3. *Given two non-empty gauges*

$$\begin{aligned} G &= [a_0 + \sum_{i=1}^n a_i \lambda_i, b_0 + \sum_{i=1}^n b_i \lambda_i] \\ G' &= [a'_0 + \sum_{i=1}^n a'_i \lambda_i, b'_0 + \sum_{i=1}^n b'_i \lambda_i] \end{aligned}$$

$G \subseteq G'$ iff $\forall i \in \{0, \dots, n\} a'_i \leq a_i \wedge b_i \leq b'_i$. The operation $G \sqcup G'$ defined as

$$[min(a_0, a'_0) + \sum_{i=1}^n min(a_i, a'_i) \lambda_i, max(b_0, b'_0) + \sum_{i=1}^n max(b_i, b'_i) \lambda_i]$$

is the least upper bound of G and G' .

It is quite intriguing that the natural order on gauges defined by the inclusion of denotations is the pointwise extension of the order on intervals. Gauges define a relational numerical domain that has the structure of a non-relational domain. This remarkable property is key to the scalability of the gauge abstraction.

Given a gauge bound g , we denote by $[g, +\infty]$ the upper gauge \overline{g} , by $[-\infty, g]$ the lower gauge \underline{g} , and by $[-\infty, +\infty]$ the trivial gauge $\mathbb{Z} \times (\mathbb{Z}^+)^n$. The order relation and the join operation defined above are readily extended to these generalized gauges, in the same way as is done for intervals. If we denote by \mathbf{G} the set of all gauges, we have established that:

Theorem 4. $(\mathbf{G}, \subseteq, \sqcup, [-\infty, +\infty])$ is a \sqcup -semilattice. The empty gauge $\perp_{\mathbf{G}}$ is the bottom element.

Note that, in general, the intersection of two gauges is not a gauge and the greatest lower bound cannot be defined.

3 Abstract Interpretation Framework

We construct our static analysis in the theoretical framework of Abstract Interpretation [8, 9]. A program is represented as a control-flow graph and operates over a set of integer variables $\mathcal{X} = \{x, y, \dots\}$ and a distinct set of integer non-negative counters $\Lambda = \{\lambda_1, \dots, \lambda_n\}$. The control-flow graph is given by a set of nodes N , an initial node $start \in N$ and a transition relation $n \rightarrow n' : cmd$ labeled by commands. A command is either a sequence $s_1; \dots; s_k$ of statements,

```

1: x = 0;
2: for (i = 0; i < 10; i++) {
3:   x += 2;
4: }
5: ...

```

$$\left\{ \begin{array}{l} 1 \rightarrow 2 : x = 0; i = 0; \mathbf{new}(\lambda) \\ 2 \rightarrow 3 : i \leq 9 \\ 3 \rightarrow 2 : x = x + 2; i = i + 1; \mathbf{inc}(\lambda, 1) \\ 2 \rightarrow 4 : 10 \leq i \\ 4 \rightarrow 5 : \mathbf{forget}(\lambda) \end{array} \right.$$

Fig. 2. Representation of a simple C program in the language of the analyzer

a condition $e_1 \leq e_2$, where e_1, e_2 are either variables or integer constants, or a counter removal operation **forget**(λ), where $\lambda \in \Lambda$. The syntax of statements is defined as follows:

$$\begin{aligned} \mathbf{stmt} ::= & x = \mathbf{exp} \quad x \in \mathcal{X} \\ & | \mathbf{new}(\lambda) \quad \lambda \in \Lambda \\ & | \mathbf{inc}(\lambda, k) \quad \lambda \in \Lambda, k \in \mathbb{Z}^+ \\ \mathbf{exp} ::= & c \quad c \in \mathbb{Z} \\ & | x \quad x \in \mathcal{X} \\ & | \mathbf{exp} + \mathbf{exp} \\ & | \mathbf{exp} - \mathbf{exp} \\ & | \mathbf{exp} * \mathbf{exp} \end{aligned}$$

The concrete semantics is defined as a transition system on a set of states Σ . A state $\sigma \in \Sigma$ is a pair $\langle n, \varepsilon \rangle$, where n is a node of the control-flow graph and $\varepsilon \in \mathbb{Z}^\mathcal{X} \times (\mathbb{Z}^+)^\Lambda$ is an environment assigning values to variables in \mathcal{X} and Λ . The semantics $\llbracket \cdot \rrbracket$ of statements and expressions is defined on environments as follows:

$$\begin{aligned} \llbracket x = e \rrbracket \varepsilon &= \varepsilon[x \mapsto \llbracket e \rrbracket \varepsilon] \\ \llbracket \mathbf{new}(\lambda) \rrbracket \varepsilon &= \varepsilon[\lambda \mapsto 0] \\ \llbracket \mathbf{inc}(\lambda, k) \rrbracket \varepsilon &= \varepsilon[\lambda \mapsto \varepsilon(\lambda) + k] \end{aligned}$$

The transition relation over states is defined as follows:

- If $n \rightarrow n' : s_1; \dots; s_k$, then $\langle n, \varepsilon \rangle \rightarrow \langle n', \llbracket s_k \rrbracket \circ \dots \circ \llbracket s_1 \rrbracket \varepsilon \rangle$,
- If $n \rightarrow n' : x \leq y$ and $\varepsilon(x) \leq \varepsilon(y)$, then $\langle n, \varepsilon \rangle \rightarrow \langle n', \varepsilon \rangle$,
- If $n \rightarrow n' : x \leq c$ and $\varepsilon(x) \leq c$, then $\langle n, \varepsilon \rangle \rightarrow \langle n', \varepsilon \rangle$ (and similarly for a constant on the left-hand side of the condition),
- If $n \rightarrow n' : \mathbf{forget}(\lambda)$, then, for any $l \in \mathbb{Z}^+$, $\langle n, \varepsilon \rangle \rightarrow \langle n', \varepsilon[\lambda \mapsto l] \rangle$.

The last rule simply expresses that the value of a counter that is removed from scope can be any nonnegative integer. An initial state in the operational semantics is a pair $\langle start, \varepsilon \rangle$, where ε is any environment, as variables are assumed to be uninitialized at the beginning of the program. We denote by \mathcal{I} the set of all initial states. Although simplified, this representation of programs is very close to the actual implementation of the analysis, which is based on LLVM [16].

In Fig. 2 we show how to translate a simple C program into our language. If the original program is structured, it is quite straightforward to introduce the

counter operations, as shown in the figure. In case the input program comes as a control-flow graph, we need to identify the loops and place the counters accordingly. This can be readily done using Bourdoncle's decomposition of a graph into a hierarchy of nested strongly connected components [2]. This efficient algorithm can be used to label each node of the control-flow graph with the sequence of nested strongly connected components in which it belongs. Using this information, loop counters can be assigned to each component and the counter operations can be automatically added to the relevant edges of the control-flow graph. The complexity of Bourdoncle's algorithm is $O(ke)$ where k is the maximum depth of loop nests and e is the number of edges in the control-flow graph.

We are interested in computing a sound approximation of the collecting semantics [6], i.e., the set of all states that are reachable from an initial state. Following the theory of Abstract Interpretation, the collecting semantics can be expressed as the least fixpoint of a semantic transformer \mathbb{F} . We denote by \mathcal{E} the set $\mathbb{Z}^X \times (\mathbb{Z}^+)^A$ of all environments. Then, the semantic transformer \mathbb{F} is the function defined over $\wp(\mathcal{E})^N$ as follows:

$$\forall n \neq start \in N : \mathbb{F}(X)(n) = \{\varepsilon \in \mathcal{E} \mid \exists n' \in N, \exists \varepsilon' \in X(n') : \langle n', \varepsilon' \rangle \rightarrow \langle n, \varepsilon \rangle\}$$

with $\mathbb{F}(X)(start) = \mathcal{I}$. In order to obtain a computable approximation of the least fixpoint $\mathbf{lfp} \mathbb{F}$, we need to construct an abstract semantic specification [9], i.e.,

- An abstract domain (D^\sharp, \sqsubseteq) together with a monotone concretization function $\gamma : (D^\sharp, \sqsubseteq) \rightarrow (\wp(\mathcal{E}), \sqsubseteq)$,
- An abstract initial state $\mathcal{I}^\sharp \in D^\sharp$ such that $\mathcal{I} \subseteq \gamma(\mathcal{I}^\sharp)$,
- An abstract semantic transformer $\mathbb{F}^\sharp : (D^\sharp)^N \rightarrow (D^\sharp)^N$ such that $\mathbb{F} \circ \gamma \subseteq \gamma \circ \mathbb{F}^\sharp$,
- A widening operator $\nabla : D^\sharp \times D^\sharp \rightarrow D^\sharp$ such that, for any sequence $(x_i^\sharp)_{i \geq 0}$ of elements of D^\sharp , the sequence $(y_i^\sharp)_{i \geq 0}$ inductively defined as:

$$\begin{cases} y_0^\sharp = x_0^\sharp \\ y_{i+1}^\sharp = y_i^\sharp \nabla x_{i+1}^\sharp \end{cases}$$

is ultimately stationary.

Then, it can be shown [9] that the sequence $(\mathbb{F}_i^\sharp)_{i \geq 0}$ iteratively defined as follows using the pointwise extension of ∇ :

$$\begin{cases} \mathbb{F}_0^\sharp = \mathcal{I}^\sharp \\ \mathbb{F}_{i+1}^\sharp = \mathbb{F}_i^\sharp \text{ if } \mathbb{F}^\sharp(\mathbb{F}_i^\sharp) \sqsubseteq \mathbb{F}_i^\sharp \\ \quad = \mathbb{F}_i^\sharp \nabla \mathbb{F}^\sharp(\mathbb{F}_i^\sharp) \text{ otherwise} \end{cases}$$

is ultimately stationary and its limit is a sound approximation of $\mathbf{lfp} \mathbb{F}$.

4 The Gauge Domain

In this section we will construct an abstract semantic specification for the gauge abstraction. We cannot use the gauge semilattice \mathbf{G} as is, because gauges are defined for all values of the counters, whereas in the first steps of the abstract iteration sequence only isolated counter values are computed. We need an operation similar to the higher-dimensional convex hull for convex polyhedra, which can build a convex approximation of a discrete set of points. In order to enable this type of induction, we need to keep track of constant counter values that are obtained at the very first steps of the abstract iteration sequence.

We denote by $(\mathbb{Z}_\top, \sqsubseteq)$ the semilattice of constants with greatest element \top . For $x, y \in \mathbb{Z}_\top$, $x \sqsubseteq y$ iff $y = \top$ or $x = y$. We define the domain of *sections* $\mathcal{S} = (\mathbb{Z}_\top^A, \sqsubseteq)$ ordered by pointwise extension of the order on \mathbb{Z}_\top . We denote by $\mathcal{E}^\sharp = (\mathbf{G}^X, \sqsubseteq)$ the set of *abstract environments* ordered by pointwise inclusion. A *gauge section* is a pair $(\rho, \varepsilon^\sharp)$, where $\rho \in \mathcal{S}$ and $\varepsilon^\sharp \in \mathcal{E}^\sharp$, such that only counters in $\rho^{-1}(\top)$ may appear inside a gauge bound of ε^\sharp . The concretization $\gamma(\rho, \varepsilon^\sharp) \in \wp(\mathcal{E})$ of the gauge section is the set of all concrete environments $\varepsilon \in \mathcal{E}$ satisfying the following property:

$$\begin{aligned} & \forall x \in X, \exists (l_1, \dots, l_n) \in (\mathbb{Z}^+)^A : (\varepsilon(x), l_1, \dots, l_n) \in \varepsilon^\sharp(x) \\ & \wedge \forall i \in \{1, \dots, n\} : \rho(\lambda_i) \neq \top \Rightarrow l_i = \rho(\lambda_i) \wedge \forall i \in \{1, \dots, n\} : \varepsilon(\lambda_i) = l_i \end{aligned}$$

A gauge section is simply an abstract environment where the value of certain counters is set. Working on gauge sections instead of gauges will allow us to construct the invariants incrementally during the abstract iteration sequence. We denote by $(\mathbf{GS}, \sqsubseteq)$ the domain of gauge sections ordered by pointwise extension of the orders on \mathcal{S} and \mathcal{E}^\sharp .

We can now construct an abstract semantic specification for the gauge abstraction. We could take \mathbf{GS} as the abstract domain of our specification. However, this choice would yield poor results on nested loops with constant iteration bounds, a very common construct in flight systems and more generally in embedded applications. In order to keep a good level of precision, we need to maintain information on the ranges of the counters. We denote by \mathbf{I} the standard lattice of intervals [7]. The abstract domain D^\sharp is given by $\mathbf{GS} \times \mathbf{I}^A$ endowed with the pointwise extension of the underlying orders. The concretization $\gamma((\rho, \varepsilon^\sharp), \ell^\sharp)$ of an element of the product domain D^\sharp is defined in the standard way as $\{\varepsilon \in \gamma(\rho, \varepsilon^\sharp) \mid \forall i \in \{1, \dots, n\} : \varepsilon(\lambda_i) \in \ell^\sharp(\lambda_i)\}$. The abstract initial state \mathcal{I}^\sharp is trivially given by the element of D^\sharp in which all components are set either to \top or to $[-\infty, +\infty]$.

The next thing we need to construct is a widening operator on D^\sharp , as it will be needed to define the abstract semantic function later on. We just need to define a widening on the domain of gauge sections, since the widening operator on D^\sharp can be obtained by pointwise application of the widenings on the underlying domains. We first need some auxiliary operations. If $G = [a_0 + \sum_{i=1}^n a_i \lambda_i, b_0 + \sum_{i=1}^n b_i \lambda_i]$ is a gauge, $j \in \{1, \dots, n\}$ and $l \in \mathbb{Z}^+$, we denote by $G[\lambda_j = l]$ the gauge

$$[a_0 + a_j l + \sum_{i \neq j} a_i \lambda_i, b_0 + b_j l + \sum_{i \neq j} b_i \lambda_i]$$

where we set the value of one counter. Let G and G' be two gauges defined as follows:

$$\begin{aligned} G &= [a_0 + \sum_{i=1}^n a_i \lambda_i, b_0 + \sum_{i=1}^n b_i \lambda_i] \\ G' &= [a'_0 + \sum_{i=1}^n a'_i \lambda_i, b'_0 + \sum_{i=1}^n b'_i \lambda_i] \end{aligned}$$

Now, assume there is $\iota \in \{1, \dots, n\}$ such that $a_\iota = b_\iota = a'_\iota = b'_\iota = 0$. Let $u, v \in \mathbb{Z}^+$ be two distinct non-negative integers. We want to construct a gauge, denoted by $G \nabla_{u,v}^{\lambda_\iota} G'$ such that

$$G[\lambda_\iota = u], G'[\lambda_\iota = v] \subseteq G \nabla_{u,v}^{\lambda_\iota} G'$$

This operation implements the basic induction step with respect to a counter. We have two gauges at two different values of the counter λ_ι and we want to extrapolate a gauge for all possible values of the counter. We choose a simple approach and perform a *linear interpolation*. We compute the slope $\alpha_\iota = \lfloor \frac{a'_0 - a_0}{v-u} \rfloor$ for the lower gauge (resp. $\beta_\iota = \lceil \frac{b'_0 - b_0}{v-u} \rceil$ for the upper gauge), taking care of rounding to the lower (resp. upper) nearest integer. This operation introduces new constants $\alpha_0 = a_0 - \alpha_\iota u$ and $\beta_0 = b_0 - \beta_\iota u$ into the gauge expression. There is no guarantee that the slopes and constants calculated from the upper (resp. lower) gauge will appear on their respective side, i.e., $\alpha_0 \leq \beta_0$ and $\alpha_\iota \leq \beta_\iota$. Therefore, we define $G \nabla_{u,v}^{\lambda_\iota} G'$ as the gauge $[c_0 + \sum_{i=1}^n c_i \lambda_i, d_0 + \sum_{i=1}^n d_i \lambda_i]$, where

- $c_0 = \min(\alpha_0, \beta_0)$
- $d_0 = \max(\alpha_0, \beta_0)$
- $c_\iota = \min(\alpha_\iota, \beta_\iota)$
- $d_\iota = \max(\alpha_\iota, \beta_\iota)$
- For $i \neq \iota$ and $i \neq 0$, $c_i = \min(a_i, a'_i)$ and $d_i = \max(b_i, b'_i)$

This elementary widening can be defined similarly when one bound of the gauges is $\pm\infty$. We need a variant of the previous operation when one of the gauges is defined over λ_ι . We keep the same notations and we now relax the assumptions, i.e., a'_ι and b'_ι may be nonzero, and $v = \top$. The gauge G' is already defined for all values of λ_ι . There is no need to change the slopes a'_ι and b'_ι , we simply need to adjust the constant coefficients. Hence, we set $\alpha_\iota = a'_\iota$ and $\beta_\iota = b'_\iota$, $\alpha_0 = a_0 - a'_\iota u$ and $\beta_0 = b_0 - b'_\iota u$. Using the previous notations, we define $G \nabla_{u,\top}^{\lambda_\iota} G'$ as the gauge $[c_0 + \sum_{i=1}^n c_i \lambda_i, d_0 + \sum_{i=1}^n d_i \lambda_i]$.

We now construct an interval-like widening ∇_I on gauges, which extrapolates unstable bounds. If we denote by L the set $\{0, \dots, n\}$, this widening is defined as follows:

$$G \nabla_I G' = \begin{cases} G & \text{if } \forall i \in L : a_i \leq a'_i \wedge b'_i \leq b_i \\ [a_0 + \sum_{i=1}^n a_i \lambda_i, +\infty] & \text{if } \exists j \in L : b_j < b'_j \wedge \forall i \in L : a_i \leq a'_i \\ [-\infty, b_0 + \sum_{i=1}^n b_i \lambda_i] & \text{if } \exists j \in L : a'_j < a_j \wedge \forall i \in L : b'_i \leq b_i \\ [-\infty, +\infty] & \text{otherwise} \end{cases}$$

Similarly, given $I \subseteq \Lambda$, we define a partial join operation \sqcup_I on gauges as follows: $G \sqcup_I G' = [\min(a_0, a'_0) + \sum_{i=1}^n \underline{a}_i \lambda_i, \max(b_0, b'_0) + \sum_{i=1}^n \bar{b}_i \lambda_i]$, where

$$\underline{a}_i = \begin{cases} \min(a_i, a'_i) & \text{if } \lambda_i \in I \\ a_i & \text{otherwise} \end{cases} \quad \text{and} \quad \bar{b}_i = \begin{cases} \max(b_i, b'_i) & \text{if } \lambda_i \in I \\ b_i & \text{otherwise} \end{cases}$$

The widening and partial join operations on gauges defined above can be extended pointwise to abstract environments in \mathcal{E}^\sharp . Now, let $(\rho_1, \varepsilon_1^\sharp)$ and $(\rho_2, \varepsilon_2^\sharp)$ be two gauge sections. Let $\Delta = \{\lambda'_1, \dots, \lambda'_k\}$ be the set of counters on which the sections ρ_1 and ρ_2 disagree, and $A = \Delta \setminus \Delta$ the set of counters on which they agree. If $\Delta \neq \emptyset$, we define the widening of the gauge sections as follows:

$$(\rho_1, \varepsilon_1^\sharp) \nabla (\rho_2, \varepsilon_2^\sharp) = \left(\rho_1 \sqcup \rho_2, \left(\cdots \left(\varepsilon_1^\sharp \nabla_{\rho_1(\lambda'_1), \rho_2(\lambda'_1)}^{\lambda'_1} \varepsilon_2^\sharp \right) \cdots \nabla_{\rho_1(\lambda'_k), \rho_2(\lambda'_k)}^{\lambda'_k} \varepsilon_2^\sharp \right) \sqcup_A \varepsilon_2^\sharp \right)$$

If $\Delta = \emptyset$, then $\rho_1 = \rho_2$, and we simply use the interval-like widening as follows:

$$(\rho_1, \varepsilon_1^\sharp) \nabla (\rho_2, \varepsilon_2^\sharp) = \left(\rho_1, \varepsilon_1^\sharp \nabla_I \varepsilon_2^\sharp \right)$$

Note that the definition of the widening depends on the order in which the counters in Δ are arranged, as the linear interpolation widening defined above is commutative but not necessarily associative. In practice, for usual loop constructs, which are the main target of our analysis, the order in which the widening operations are performed has no effect on the result, but this may not always be the case. This is one limitation of our approach as compared to convex polyhedra and weakly relational domains.

We are now ready to define the abstract semantic function \mathbb{F}^\sharp . We first define the abstract semantics of expressions. Let G and G' be two gauges defined as follows:

$$\begin{aligned} G &= [a_0 + \sum_{i=1}^n a_i \lambda_i, b_0 + \sum_{i=1}^n b_i \lambda_i] \\ G' &= [a'_0 + \sum_{i=1}^n a'_i \lambda_i, b'_0 + \sum_{i=1}^n b'_i \lambda_i] \end{aligned}$$

We define $G + G' = [(a_0 + a'_0) + \sum_{i=1}^n (a_i + a'_i) \lambda_i, (b_0 + b'_0) + \sum_{i=1}^n (b_i + b'_i) \lambda_i]$ and $G - G' = [(a_0 - b'_0) + \sum_{i=1}^n (a_i - b'_i) \lambda_i, (b_0 - a'_0) + \sum_{i=1}^n (b_i - a'_i) \lambda_i]$. Since the gauge abstraction is linear, we cannot compute the multiplication exactly. In practice, multiplication mostly occurs in pointer arithmetic when scaling a byte offset to fit a type of a certain size. Hence, it is sufficient to consider the case when one of the gauges is a singleton, say $G' = [c, c]$. Then we define

$$G * G' = \left[ca_0 + \sum_{i=1}^n c a_i \lambda_i, cb_0 + \sum_{i=1}^n c b_i \lambda_i \right]$$

if c is positive, swapping the bounds when c is negative. Other cases when G is constant, both gauges are constant or one is zero are handled similarly. In all other cases we just return the trivial gauge $[-\infty, +\infty]$. For brevity, we did not go over the cases when one of the gauge bounds is infinite as they are handled similarly. The abstract semantics of expressions is readily defined from the previous operations on gauges.

Now, let $((\rho, \varepsilon^\sharp), \ell^\sharp)$ be an element of D^\sharp . We define the abstract semantics of statements as follows. In the case of an assignment operation, we have

$$[x = e]^\sharp((\rho, \varepsilon^\sharp), \ell^\sharp) = ((\rho, \varepsilon^\sharp[x \mapsto [e]^\sharp \varepsilon^\sharp]), \ell^\sharp)$$

For any counter λ , we denote by $\varepsilon^\sharp|_\lambda$ the abstract environment in which all occurrences of a gauge where λ appears with a non-zero coefficient have been

replaced with $[-\infty, +\infty]$. Then, the abstract semantics of a **new**(λ) operation can be defined as follows:

$$\llbracket \mathbf{new}(\lambda) \rrbracket^\sharp((\rho, \varepsilon^\sharp), \ell^\sharp) = ((\rho[\lambda \mapsto 0], \varepsilon^\sharp|_\lambda), \ell^\sharp[\lambda \mapsto [0, 0]])$$

Given a gauge $G = [a_0 + \sum_{i=1}^n a_i \lambda_i, b_0 + \sum_{i=1}^n b_i \lambda_i]$, a counter λ_j and $k \in \mathbb{Z}^+$, we define the gauge $inc_{\lambda_j, k}(G)$ as follows:

$$\left[\min(a_0 - ka_j, a_0 - kb_j) + \sum_{i=1}^n a_i \lambda_i, \max(a_0 - ka_j, a_0 - kb_j) + \sum_{i=1}^n b_i \lambda_i \right]$$

This operation corresponds to incrementing a counter by a constant. The resulting constant coefficients may not satisfy the consistency condition for a non-empty gauge, whence the introduction of *min* and *max* operations. Cases where one of the gauge bounds is infinite are handled similarly. We can extend this operation pointwise to abstract environments. Thus, we can define the semantics of a **inc**(λ, k) operation as follows:

$$\llbracket \mathbf{inc}(\lambda, k) \rrbracket^\sharp((\rho, \varepsilon^\sharp), \ell^\sharp) = ((\rho[\lambda \mapsto \rho(\lambda) + k], inc_{\lambda, k}(\varepsilon^\sharp)), \ell^\sharp[\lambda \mapsto \ell^\sharp(\lambda) + k])$$

Note that for clarity we have overloaded the addition operator, but its semantics depends on the domain on which it applies.

It now remains to define the abstract semantics of commands. For a sequence of statements $s_1 \dots s_n$, the abstract semantics is obviously given by $\llbracket s_n \rrbracket^\sharp \circ \dots \circ \llbracket s_1 \rrbracket^\sharp$. The abstract semantics of a condition $x \leq y$ is defined as follows. Assume that $a_0 + \sum_{i=1}^n a_i \lambda_i$ is the lower gauge bound of $\varepsilon^\sharp(x)$ and $b_0 + \sum_{i=1}^n b_i \lambda_i$ is the upper gauge bound of $\varepsilon^\sharp(y)$. We denote by C the linear inequality constraint $a_0 - b_0 + \sum_{i=1}^n (a_i - b_i) \lambda_i \leq 0$. Then we define

$$\llbracket x \leq y \rrbracket^\sharp((\rho, \varepsilon^\sharp), \ell^\sharp) = ((\rho, \varepsilon^\sharp), reduce_C(\ell^\sharp))$$

where $reduce_C(\ell^\sharp)$ is the reduction of a collection of variable ranges against a linear inequality constraint, using the algorithm defined in [13]. Since this algorithm is based on constraint propagation, we arbitrarily limit the number of propagation cycles performed (the threshold in our implementation is 5) so as to maintain an $O(|\Lambda|)$ complexity. No impact on precision has been observed in our experiments. The other types of conditions are handled similarly. Note that this operation only affects the loop counter bounds and does not change the gauge invariants.

Now, consider a gauge $G = [a_0 + \sum_{i=1}^n a_i \lambda_i, b_0 + \sum_{i=1}^n b_i \lambda_i]$, a counter λ_j and an interval $[l, u]$. We define the operation $coalesce_{\lambda_j, [l, u]}(G)$ as follows:

$$coalesce_{\lambda_j, [l, u]}(G) = \left[a_0 + a_j l + \sum_{i \neq j} a_i \lambda_i, b_0 + b_j u + \sum_{i \neq j} b_i \lambda_i \right]$$

We can extend this operation pointwise on abstract environments. Then, we can define the semantics of the **forget**(λ) operation as follows:

$$\llbracket \mathbf{forget}(\lambda) \rrbracket((\rho, \varepsilon^\sharp), \ell^\sharp) = ((\rho[\lambda \mapsto \top], coalesce_{\lambda, \ell^\sharp(\lambda)}(\varepsilon^\sharp)), \ell^\sharp[\lambda \mapsto [-\infty, +\infty]])$$

The **forget**(λ) operation is used when exiting the scope of a loop. If we did not inject the range information of the loop counter back into the gauge invariants, we would incur a major loss of accuracy when analyzing loops with constant iteration bounds. This points to a major limitation of the gauge abstraction: it only maintains precise loop invariants *inside* a loop, but most of this information is lost when exiting the loop. The polyhedral domain keeps relational information across loop boundaries and is more precise in this respect.

Finally, we define the abstract semantic transformer \mathbb{F}^\sharp as follows

$$\forall n \neq start \in N : \mathbb{F}^\sharp(X)(n) = \nabla\{\llbracket cmd \rrbracket^\sharp(X(n')) \mid n' \rightarrow n : cmd\}$$

with $\mathbb{F}^\sharp(X)(start) = \mathcal{I}^\sharp$. Note that the widening operation is used to merge the invariants over a join node. We only need to use the interval-like widening ∇_I and the widening on I^A when it is the entry node of a strongly connected component, otherwise we can simply use the join operations, which provide better accuracy.

All elementary domain operations only depend on the number of active loop counters and the number of variables in the program. Using a sparse implementation of abstract environments, it is not difficult to see that all operations have an $O(km)$ time complexity in the worst case, where m is the number of program variables and k is the maximum depth of loop nests in the program. If we consider k as a constant, which is a realistic assumption in practice, all operations are linear in the number of program variables. The gauge domain has a very low complexity in the worst case and is guaranteed to scale for large programs.

In order to illustrate how the abstract semantics operates, we unroll the first few steps of the abstract iteration sequence on the program shown in Fig. 2.

- Node 1: $((\{\}, \{\}), \{\})$
- Node 2:

$$\left(\left(\{\lambda \mapsto 0\}, \left\{ \begin{array}{l} x \mapsto [0, 0] \\ i \mapsto [0, 0] \end{array} \right\} \right), \{\lambda \mapsto [0, 0]\} \right)$$

- Node 3: The reduction operation has no effect on the invariant

$$\left(\left(\{\lambda \mapsto 0\}, \left\{ \begin{array}{l} x \mapsto [0, 0] \\ i \mapsto [0, 0] \end{array} \right\} \right), \{\lambda \mapsto [0, 0]\} \right)$$

- Node 2 through the back edge:

$$\left(\left(\{\lambda \mapsto 1\}, \left\{ \begin{array}{l} x \mapsto [2, 2] \\ i \mapsto [1, 1] \end{array} \right\} \right), \{\lambda \mapsto [1, 1]\} \right)$$

We perform the linear interpolation widening and we obtain:

$$\left(\left(\{\lambda \mapsto \top\}, \left\{ \begin{array}{l} x \mapsto [2\lambda, 2\lambda] \\ i \mapsto [\lambda, \lambda] \end{array} \right\} \right), \{\lambda \mapsto [0, +\infty]\} \right)$$

This is the limit and convergence will be confirmed at the next iteration.

- Node 3: we perform the reduction operation on intervals and we obtain

$$\left(\left(\{\lambda \mapsto \top\}, \left\{ \begin{array}{l} x \mapsto [2\lambda, 2\lambda] \\ i \mapsto [\lambda, \lambda] \end{array} \right\} \right), \{\lambda \mapsto [0, 9]\} \right)$$

The information on the loop bounds has been recovered thanks to the reduction operation.

Analysis	Analysis Time	Precision
Intervals + Complete Inlining	41 min	79%
Commercial Tool	5 hours	91%
Octagons	> 27 hours	N/A
Gauges	10 min 30 sec	91%

Fig. 3. Experimental results

5 Experimental Evaluation

We have implemented the gauge domain described in this paper in a buffer-overflow analyzer for C programs. The gauge domain is well suited for this kind of application, as it is good at discovering invariants that hold inside usual loop constructs. The buffer-overflow analyzer is implemented within an Abstract Interpretation framework developed at NASA Ames Research Center by the author and named IKOS (Inference Kernel for Open Static Analyzers). It is beyond the scope of this paper to describe the design of the buffer-overflow analyzer. We can just say that it is based on the LLVM front-end [16] and computes an abstract representation of objects and pointers in a C program. The analysis is modular and the effect of each function in memory is summarized by numerical constraints on array indices and pointer offsets that are affixed to the abstract memory graph. These numerical constraints are represented by gauges. Symbolic bounds (such as the size of an array passed as an argument to a function) are represented using an elementary domain of symbolic constants, which is used in combination with the gauge abstraction.

We have run the analyzer on a large flight system developed at NASA Dryden Flight Research Center and Ames Research Center. It consists of 144 KLOC of C and implements advanced adaptive avionics for intelligent flight control. It is a very pointer intensive application where matrix operations are pervasive. We have compared the performance of this analyzer with that of (1) a simple interval analysis running on a version of the program where function calls have been completely expanded using the LLVM inliner, (2) a leading commercial static analyzer based on Abstract Interpretation, and (3) a version of our analyzer in which octagons [19] have been substituted for gauges. In the latter, we used Miné’s implementation of the octagon domain from the APRON library [14]. The results of these experiments are presented in Fig. 3. All analyzers ran on a MacBook Air with a 1.86 Ghz Intel Core 2 Duo and 2 GB of memory, except the commercial tool, which is installed on a high-end server with 32 CPU cores and 64 GB of memory. The precision denotes the fraction of all array-bound operations which could be statically verified by the analyzer. This figure is not available for the version of our analyzer based on octagons, as we decided to kill the analysis process after allowing it to run continuously for over 27 hours.

6 Conclusion

We have constructed a numerical relational domain that is able to infer precise loop invariants and is guaranteed to scale thanks to tight bounds on the complexity of the domain operations. An experimental study led on a complex flight system developed at NASA showed that the gauge abstraction is able to deliver accurate loop invariants in a consistent way. This domain is not intended to be a replacement for more costly relational domains like convex polyhedra. It should be seen as a cheap numerical analysis that is able to discharge many simple verification properties, so that more powerful and computationally costly domains can be used to focus on a significantly smaller portion of the program.

Acknowledgement. We are extremely grateful to Tim Reyes for spending many hours getting the code through the commercial static analyzer.

References

1. Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Ghorbal, K., Goubault, E., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., Turin, M.: Space software validation using abstract interpretation. In: Proc. of the International Space System Engineering Conference, Data Systems in Aerospace (DASIA 2009), pp. 1–7 (2009)
2. Bourdoncle, F.: Efficient Chaotic Iteration Strategies with Widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 46–55. Springer, Heidelberg (1993)
3. Brat, G., Venet, A.: Precise and scalable static program analysis of NASA flight software. In: Proc. of the IEEE Aerospace Conference (2005)
4. Chernikova, N.V.: Algorithm for discovering the set of all the solutions of a linear programming problem. U.S.S.R. Computational Mathematics and Mathematical Physics 8(6), 282–293 (1968)
5. Clarisó, R., Cortadella, J.: The Octahedron Abstract Domain. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 312–327. Springer, Heidelberg (2004)
6. Cousot, P.: Semantic foundations of program analysis. In: Program Flow Analysis: Theory and Applications, ch. 10, pp. 303–342. Prentice-Hall (1981)
7. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. of the International Symposium on Programming (ISOP 1976), pp. 106–130 (1976)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the Symposium on Principles of Programming Languages (POPL 1977), pp. 238–252 (1977)
9. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ Analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of the Symposium on Principles of Programming Languages (POPL 1978), pp. 84–97 (1978)

12. Dax, A.: An elementary proof of Farkas' lemma. *SIAM Rev.* 39(3), 503–507 (1997)
13. Harvey, W., Stuckey, P.: Improving linear constraint propagation by changing constraint representation. *Constraints* 8(2), 173–207 (2003)
14. Jeannet, B., Miné, A.: APRON: A Library of Numerical Abstract Domains for Static Analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
15. Lviron, V., Logozzo, F.: SubPolyhedra: A (More) Scalable Approach to Infer Linear Inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
16. The LLVM Compiler Infrastructure, <http://llvm.org>
17. Logozzo, F., Fähndrich, M.: Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: Proc. of the ACM Symposium on Applied Computing (SAC 2008), pp. 184–188 (2008)
18. Miné, A.: A New Numerical Abstract Domain Based on Difference-Bound Matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
19. Miné, A.: The octagon abstract domain. In: Proc. of the Workshop on Analysis, Slicing, and Transformation (AST 2001), pp. 310–319 (2001)
20. Miné, A.: A Few Graph-Based Relational Numerical Abstract Domains. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 117–132. Springer, Heidelberg (2002)
21. Motzkin, T.S., Raiffa, H., Thompson, G.L., Thrall, R.M.: The double description method. *Annals of Mathematics Studies* II(28), 51–73 (1953)
22. Sankaranarayanan, S., Colón, M.A., Sipma, H.B., Manna, Z.: Efficient Strongly Relational Polyhedral Analysis. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 111–125. Springer, Heidelberg (2005)
23. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program Analysis Using Symbolic Ranges. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 366–383. Springer, Heidelberg (2007)
24. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
25. Seidl, H., Flexeder, A., Petter, M.: Interprocedurally Analysing Linear Inequality Relations. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 284–299. Springer, Heidelberg (2007)
26. Simon, A., King, A.: Exploiting Sparsity in Polyhedral Analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 336–351. Springer, Heidelberg (2005)
27. Simon, A., King, A., Howe, J.M.: Two Variables per Linear Inequality as an Abstract Domain. In: Logic-Based Program Synthesis and Transformation, pp. 71–89 (2003)
28. Venet, A., Brat, G.P.: Precise and efficient static array bound checking for large embedded C programs. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2004), pp. 231–242 (2004)
29. Ziegler, G.M.: Lectures on Polytopes. Graduate Texts in Mathematics. Springer (1995)

Diagnosing Abstraction Failure for Separation Logic–Based Analyses

Josh Berdine¹, Arlen Cox^{2,*}, Samin Ishtiaq¹, and Christoph M. Wintersteiger¹

¹ Microsoft Research, Cambridge

² University of Colorado, Boulder

Abstract. Abstraction refinement is an effective verification technique for automatically proving safety properties of software. Application of this technique in shape analyses has proved impractical as core components of existing refinement techniques such as backward analysis, general conjunction, and identification of unreachable but doomed states are computationally infeasible in such domains.

We propose a new method to diagnose proof failures to be used in a refinement scheme for Separation Logic–based shape analyses. To check feasibility of abstract error traces, we perform Bounded Model Checking over the traces using a novel encoding into SMT. A subsequent diagnosis finds discontinuities on infeasible traces, and identifies doomed states admitted by the abstraction. To construct doomed states, we give a model-finding algorithm for “symbolic heap” Separation Logic formulas, employing the execution machinery of the feasibility checker to search for concrete counter-examples. The diagnosis has been implemented in SLAYER, and we present a simple scheme for refining the abstraction of hierarchical data structures, and illustrate its effectiveness on benchmarks from the SLAYER test suite.

1 Introduction

Abstraction refinement has proven to be an effective technique for verification of safety properties of software. Iterative refinement of the abstraction allows the use of a coarse and computationally cheap abstraction that often suffices to prove the desired property. If the abstraction is not precise enough, it supports incremental shifting to a potentially very precise and computationally expensive analysis. This technique has been very successfully applied to predicate abstraction domains. Not so for shape analyses. The consequence is that the abstractions used in shape analyses must be very conservative, since any information that is abstracted away is forever irrecoverable. One solution is to simply choose the right abstraction in the first place, but while this can be computationally efficient, the choice is sensitive to the property and program, making this approach difficult to use in tools intended to be somewhat generally applicable.

To explain why a straightforward analogue of traditional counter-example guided abstract refinement (CEGAR) techniques used for predicate abstraction

* This work was performed while an intern at Microsoft Research, Cambridge.

does not work for shape analyses, recall a basic CEGAR procedure. Suppose that the goal is to prove that some error state is not reachable, and that for a given abstraction this proof fails. In this context it is common to abstract traces rather than just states, and failing to find a proof amounts to finding an abstract trace to error, t . The first question is whether t constitutes a disproof of the property, or witnesses that the abstraction is too coarse to prove the property. This question can be answered by checking feasibility of t , that is, whether or not it represents at least one concrete trace. If so, there is nothing to do; the concrete trace witnesses that the program violates the property. If t is not feasible, then it must contain a *discontinuity* where concrete execution cannot follow the abstract trace. That is, for every concrete trace along t from an initial state to a state s at the discontinuity, execution would have to “leap sideways” to some unreachable state s' that the abstraction does not distinguish from s , before concrete execution from s' may proceed to reach error. The aim of abstraction refinement is to increase the precision of the abstraction in order to partition the *doomed* states such as s' from the others, and thereby avoid the introduction of t . To perform an effective refinement, a discontinuity and a characterization of doomed states is found, that is, the failure of abstraction is *diagnosed*. One way to do so is to search for a program point ℓ on t such that the over-approximation Q of the reachable states after executing along t to ℓ and the weakest precondition with respect to error of the command C along the suffix of t from ℓ to *error*, $wp(C, \text{error})$, are consistent, i.e., $Q \wedge wp(C, \text{error}) \neq \text{false}$.

In this case ℓ is a discontinuity and the models of the formula $Q \wedge wp(C, \text{error})$ are doomed states that need to be partitioned from others. There are various refinement techniques, but the use of precondition computation and conjunction or similar operations is ubiquitous.

The use of precondition computation and conjunction presents a serious problem in the context of shape analysis. To get an understanding of why backward shape analysis is very expensive, consider the weakest precondition of a command that swings a pointer stored at x from one object to another p resulting in a state satisfying Q : $P = wp(*x = p, Q)$. In the states that satisfy P , there are many possible aliasing configurations for x , and $*x$ might point to any object at all, or be any dangling pointer. There are very many such states, and they are not uniform in a way for which known shape analysis domains provide compact representations. Additionally, shape analyses based on separation logic use “symbolic heap” fragments of the logic similar to that introduced by Smallfoot [4], which do not include general conjunction. Reducing a general conjunction to a symbolic heap formula is theoretically possible, but computationally infeasible.

Therefore, an abstraction failure diagnosis that avoids precondition computation and general conjunction is a prerequisite for refinement of shape abstractions in a fashion similar to that applied for predicate abstraction. We propose to refine based on individual doomed states introduced by abstraction, rather than symbolic representations of all such states, and present a diagnosis technique that identifies discontinuities on abstract traces obtained from failed separation logic proofs and fabricates doomed states showing where the abstraction is too coarse.

Our procedure starts with a failed separation logic proof in the form of an abstract transition system and slices out an abstract counter-example. These abstract counter-examples generally contain loops and hence represent infinitely-many abstract error traces. A finite subset of these abstract traces is checked for feasibility using a very precise modeling of memory allocation and a new technique for encoding bounded model checking (BMC) as a single satisfiability modulo theories (SMT) problem, using quantified formulas with uninterpreted functions and bit-vectors.

If a concrete counter-example is not found, then a new algorithm is used to diagnose the failure of abstraction. This proceeds by searching through the points on the abstract counter-example for a discontinuity. For each point on the abstract counter-example, the prefix leading to the point is replaced with code that generates concrete states represented by the abstract state at that point. If this new abstract counter-example is feasible, then the program point under consideration contains a discontinuity, and the generated state is doomed. The diagnosis algorithm reports the input and output of abstraction and the doomed state witnessing that the abstraction was too coarse.

It should be emphasized that the state-of-the-art in refinement of shape abstractions is manual. When a shape analysis fails, the reason must be diagnosed by hand, and the definition of abstraction must be changed by hand. As the size of analyzed programs increases, the time and effort involved in diagnosing abstraction failure becomes a practical bottleneck. Therefore, automatic diagnosis of abstraction failure by itself represents a significant advance. Additionally, as a demonstration and quality check of the diagnosis, we present a simple automatic abstraction refinement scheme which uses the discontinuity and doomed state to select which “patterns” to use for abstracting hierarchical data structures.

2 Separation Logic-Based Shape Analysis

Before presenting the material on abstraction failure diagnosis, we must provide some background on shape analysis using separation logic. In particular we introduce programs, abstract states, abstract transition systems, failed proofs, and give some description of pattern-based abstraction.

Programs. Assuming some language of pure *expressions* E , the language of state-transforming *commands* is generated by the following grammar:

$$\begin{aligned}
 C ::= & x = \text{malloc}(E) \mid \text{free}(x) && \text{allocate and delete heap memory} \\
 & \mid x = \text{nondet}() \mid x = E && \text{kill and move (register)} \\
 & \mid *x = y \mid x = *y && \text{store and load (heap)} \\
 & \mid \text{assume}(E) \mid \text{assert}(E) && \text{assumptions and assertions} \\
 & \mid \text{nop} \mid C; C && \text{sequential composition}
 \end{aligned}$$

A *program* is defined by its control-flow graph $\langle \mathbb{L}, \mathbb{E}, \ell_0, \kappa \rangle$, where the vertices \mathbb{L} are program locations, the program entry point is the root $\ell_0 \in \mathbb{L}$, and the edges $\mathbb{E} \subseteq \mathbb{L} \times \mathbb{L}$ are labeled with commands by $\kappa : \mathbb{E} \rightarrow C$.

Abstract States. Separation logic-based shape analyses represent sets of program states using formulas in a “symbolic heaps” fragment [4] of separation logic’s assertion language [24]. Our diagnosis algorithms are implemented in SLAYER [5], which uses the following language of *formulas*:

$$\begin{aligned} P, Q ::= & F && \text{first-order formulas} \\ | \mathsf{emp} & | l \mapsto r & | \mathsf{ls}(\Lambda, k, \mathbf{p}, \mathbf{f}, \mathbf{b}, \mathbf{n}) & \text{atomic heap formulas} \\ | P * Q & | P \vee Q & | \exists \mathbf{x}. Q \end{aligned}$$

Apart from emp , which describes the empty part of a heap, atomic heap formulas are of two forms: points-to or list-segment. A points-to $l \mapsto r$ describes a single heap object at location l that contains a value described by record r . A list-segment $\mathsf{ls}(\Lambda, k, \mathbf{p}, \mathbf{f}, \mathbf{b}, \mathbf{n})$ describes a possibly-empty, possibly-cyclic, segment of a doubly-linked list, where the heap structure of each item of the list is given by Λ . In particular, $\mathsf{ls}(\Lambda, k, \mathbf{p}, \mathbf{f}, \mathbf{b}, \mathbf{n})$ is the least predicate satisfying

$$\begin{aligned} \mathsf{ls}(\Lambda, k, \mathbf{p}, \mathbf{f}, \mathbf{b}, \mathbf{n}) \text{ iff } & (k = 0 \wedge \mathbf{f} = \mathbf{n} \wedge \mathbf{p} = \mathbf{b}) \\ & \vee \exists \mathbf{x}', \mathbf{y}'. k > 0 \wedge \Lambda(\mathbf{p}, \mathbf{f}, \mathbf{x}', \mathbf{y}') * \mathsf{ls}(\Lambda, k-1, \mathbf{x}', \mathbf{y}', \mathbf{b}, \mathbf{n}) . \end{aligned}$$

See [3] for details on this predicate, but note that $\mathbf{p}, \mathbf{f}, \mathbf{b}, \mathbf{n}$ denote vectors of parameters, which are sometimes empty and written simply as a space.

The set of formulas is closed under separating conjunction $P * Q$, disjunction $P \vee Q$, and existential quantification $\exists \mathbf{x}. Q$. Note the absence of conjunction and negation of heap formulas. The pure, heap-independent, part of the logic (F) is essentially passed through to the Z3 SMT solver [17]. We assume that first-order formulas are among the expressions, $F \subset E$.

The set of *abstract states* is Q^\top , where \top is the error state.

Pattern-Based Abstraction. The abstraction performed by SLAYER is parameterized by “patterns”, the Λ argument formulas of the ls predicate that describe the shape of hierarchical data structures. See [3] for more detail, but as an example, consider a pattern for simple singly-linked lists

$$_\mathsf{SLL_ENTRY}(, front, , next) = (front \mapsto [\mathsf{Flink}: next])$$

and a pattern for singly-linked lists where each item carries a data object

$$_\mathsf{SLL_OJBS}(, front, , next) = \exists d', r. (front \mapsto [\mathsf{Data}: d'; \mathsf{Flink}: next]) * (d' \mapsto r) .$$

Abstracting the formula, which represents a list of two items carrying data,

$$\begin{aligned} \exists d'_0, d'_1, f', r_0, r_1. & (\mathsf{head} = item) \wedge (nd \neq 0) * \\ & (\mathsf{head} \mapsto [\mathsf{Data}: d'_0; \mathsf{Flink}: f']) * (d'_0 \mapsto r_0) * \\ & (f' \mapsto [\mathsf{Data}: d'_1; \mathsf{Flink}: 0]) * (d'_1 \mapsto r_1) \end{aligned}$$

using `_SLL_ENTRY` results in (a warning about leaked memory and)

$$\exists l . (head = item) \wedge (nd \neq 0) * \text{ls}(\text{_SLL_ENTRY}, l, , head, , 0)$$

while using `_SLL_OBJS` results in

$$\exists l . (head = item) \wedge (nd \neq 0) * \text{ls}(\text{_SLL_OBJS}, l, , head, , 0).$$

Note that abstracting using `_SLL_OBJS` produces a logically stronger result than abstracting using `_SLL_ENTRY`. The former preserves the fact that the Data fields point to valid objects, while the latter loses this information. As a result, adjusting the patterns used for abstraction provides an effective mechanism for abstraction refinement, analogous to the set of predicates used to control precision of predicate abstraction.

Abstract Transition Systems and Failed Proofs. SLAYER abstracts a program to an *abstract transition system* (ATS). An ATS is a graph $\langle \mathbb{L}, \mathbb{E}, \ell_0, \kappa, \delta \rangle$, which is a program where program points are labeled with abstract states by $\delta : \mathbb{L} \rightarrow Q^\top$.

An ATS is constructed by the analysis while exploring the computation tree of the program under the abstract semantics, creating cycles when abstract states are covered by existing ones. A fully-expanded ATS where no vertex is labeled with \top induces a *proof* in separation logic, where for each edge $e = (\ell_i, \ell_j) \in \mathbb{E}$, the triple $\{\delta\ell_i\} \kappa e \{\delta\ell_j\}$ is valid.

An ATS where some vertex ℓ_e is labeled with \top constitutes a *failed proof*. If $\delta\ell_e = \top$, then ℓ_e is an *error* vertex, and the ATS restricted to the transitive predecessors of ℓ_e is an *abstract counter-example*. An abstract counter-example is either concretely feasible, or it witnesses that the abstraction is too coarse.

Abstract Programs. The CEGAR approach to model checking commonly involves construction of an abstract program. If the abstract program contains an error, subsequent analysis finds an abstract trace that shows it. If this trace is infeasible in the abstract program, then it is also infeasible in the concrete program, and refinement may be performed based on the explanation for abstract infeasibility. If it is feasible in the abstract program, then it is checked for feasibility in the concrete program to determine whether it corresponds to a concrete error or should be refined.

We do not use such a two-staged approach. While we do employ abstraction functions to obtain an ATS, they are used to abstract sets of program states, producing *abstract states*, instead of directly abstracting program transitions, producing *abstract transitions*. The ATS is therefore a relation over abstract states, *not* an abstracted relation over concrete states. An abstract program could be obtained from the ATS, however, *all* error traces in the ATS will be feasible in the resulting system.

In short, since we do not use a postcondition computation that loses more precision than required by the abstraction,¹ there is no need to check if a potential counter-example is due to imprecision in the postcondition computation.

Aside: Some theoretical results regarding the complexity of adding arbitrary Boolean connectives to the fragments of separation logic used in analyzers are known [10]. For the simple propositional case with no inductive definitions, the model checking problem is NP-complete and the validity problem is Π_2^P -complete. Adding general Boolean conjunction preserves these bounds. General negation is more problematic, both problems become PSPACE-complete even in this simple case. Furthermore, performing backward analysis in the known way requires \neg^* [24], which also brings both problems up to PSPACE-complete.]

3 Abstraction Failure Diagnosis

Our approach to failure diagnosis is meant to be employed in the context of abstraction refinement. We therefore give a brief overview using a typical abstraction refinement algorithm for presentation purposes. Algorithm 1 first runs an abstract interpreter in `analyze` and if it succeeds, it returns `Safe`. If not, we simplify ATS using `slice` and then search for a concrete counter-example via `feasible`, which is described in Section 4. If it has a concrete counter-example, then we report `Unsafe`. If it does not have a concrete counter-example, we try to refine the abstraction. The `diagnose` procedure searches for doomed states and is described in Section 5. It returns a description of the discontinuity at which the abstract state was identified as doomed. If such a state is not found, or if the refinement fails for other reasons, the algorithm terminates with a result of `PossiblyUnsafe`. Otherwise it repeats the process using the new abstraction.

We illustrate the behavior of our algorithm on the simple linked list program depicted in Figure 1. This program creates a list of non-deterministic length with a heap allocated data object in every element and then deletes the list. This program is safe.

SLAYER initially fails to prove that the program is safe, the corresponding ATS is shown in Figure 2(a). At the transition from vertex 4 to vertex 3, the abstract interpreter explored the first while loop twice, creating and explicitly tracking a list of length 2 with points-to predicates. At the third iteration of the loop, it widens at vertex 2. In doing so, it selects an `_SLL_ENTRY` shape, thereby discarding information that is required to complete the proof. It still has a d_0 data object, but it has lost d_1 and it has lost any connection between data elements like d_0 and the list itself. When the abstract interpreter reaches the last command through the transition from vertex 1 to 0, it no longer knows if the particular list element points to the beginning of allocated memory or not. As a result, the proof attempt fails.

Once `analyze` terminates with a failed proof attempt, `feasible` attempts to find a concrete counter-example in the abstract counter-example. Since this program is safe, it does not find one, and the algorithm then runs `diagnose` which

¹ With the exception of losing some disequations between deallocated addresses.

Algorithm 1. Abstraction refinement algorithm

```

let abstraction_refinement prog abstraction =
  let ats = analyze prog abstraction in
  if safe ats
    return Safe
  else
    let abstract_cex = slice ats in
    let concrete_cex = feasible abstract_cex bound in
    if concrete_cex != None
      return Unsafe
    else
      let failure = diagnose abstract_cex bound in
      if failure = None
        return PossiblyUnsafe
      else
        let abstraction' = refine failure abstraction in
        if abstraction' = None
          return PossiblyUnsafe
        else
          return abstraction_refinement prog abstraction'

```

searches for a concrete counter-example starting from each widened state in the abstract counter-example. In this example, the state at vertex 2 is the only widened state. It then synthesizes a new, temporary ATS shown in Figure 2(b) which is constructed to generate all models of the separation logic formula on the vertex (within bounds). It then continues to check feasibility of counter-examples in this new ATS, which, in this example, yields a counter-example that constructs a single element list, where the data pointer is invalid.

Now that a doomed state has been found, the `refine` procedure attempts to construct a more precise abstraction. It succeeds only if it is able to find a new abstraction in which the doomed state is no longer included at the discontinuity. In this example, the `refine` procedure implemented in SLAYER (see Section 6) activates the previously inactive `_SLL_OBJS` pattern which preserves information about the `Data` objects. Finally, it restarts the abstract interpreter with the new abstraction, which, in this example, is successful in proving safety of the program.

4 Feasibility Checking

When the abstract interpretation is unable to show that a program is safe, we obtain an ATS which represents the relevant parts of the program together with an abstract model (abstract values for every variable at every control location). To distinguish between actual errors and abstraction failures, we check feasibility of error traces in the ATS. Note that this is a general verification problem and that we may employ any of a multitude of Model Checking algorithms to solve

```

1 typedef struct _SLL_ENTRY {
2   void* Data;
3   struct _SLL_ENTRY *Flink;
4 } SLL_ENTRY, *PSLL_ENTRY;
5
6 void main(void) {
7   SLL_OBJS *head = NULL, *item;
8   while (nondet()) {
9     item = (PSLL_ENTRY)malloc(sizeof(SLL_ENTRY));
10    item->Data = (int*)malloc(sizeof(int));
11    item->Flink = head;
12    head = item;
13 }
14   while (head) {
15     item = head;
16     head = item->Flink;
17     free(item->Data);
18     free(item);
19   }
20 }
```

Fig. 1. An example program

this problem. Here, we propose a Bounded Model Checker (BMC). For any fixed unrolling depth, this represents an under-approximation of the ATS. The trade-off between precision and efficiency is of paramount importance in practice and we propose to use BMC because it conveniently offers fine-grained control over the precision through a single parameter.

Recent advances in SMT solving have made it possible to encode BMC instances through a single query to the theorem prover [25] and to solve them by providing efficient quantifier instantiation and elimination procedures. In particular, the theory of bit-vectors with uninterpreted functions and quantifiers (SMT UFBV) has been shown to be a very effective means of analyzing BMC instances [33]. This theory allows for an encoding that does not require a pre-determined unrolling depth for every loop, but for the whole system, i.e., the unrolling bound corresponds to the number of nodes visited in the ATS, but the SMT solver may freely chose a different bound for each loop in the ATS. This simplifies the analysis and allows the utilization of powerful heuristics employed by SMT solvers to increase performance.

4.1 A Memory Model

To encode an ATS into SMT UFBV, a memory model is required. To achieve maximum precision, we use a flat memory model that implements accurate execution semantics. A segmented model might be easier to analyze, but would introduce unsoundness [18].

This choice is motivated by the particular interest in detecting four specific classes of errors: 1) Array out of bounds errors; 2) Dereferencing NULL pointers; 3) Double frees; and 4) Frees of unallocated memory. In a flat memory model, these four errors can be reduced to two: out of bounds errors and NULL pointer errors can both be treated as dereferencing unallocated memory; a double free error corresponds to an attempt to free unallocated memory.

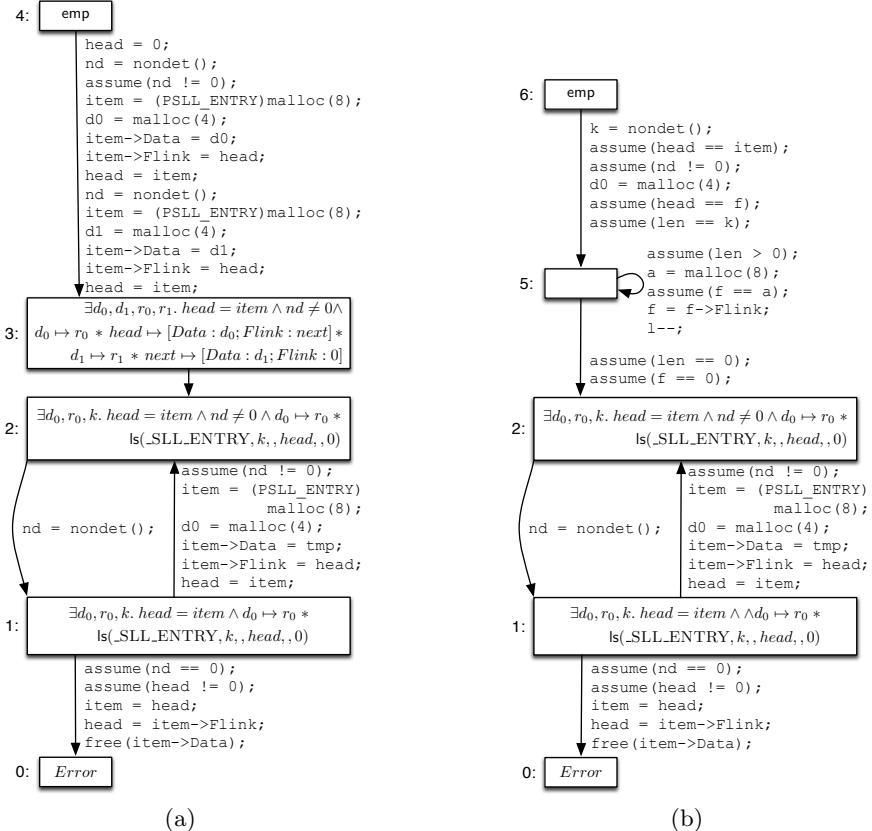


Fig. 2. (a) Abstract counter-example prior to refinement and (b) with prefix of vertex 2 replaced

Memory allocation must be modeled accurately for a flat model to be able to find errors. If a strategy is chosen similar to a real memory allocator (first fit, best fit, etc), the objects are packed together and will likely not cause errors when accessing out of bounds array elements. For this reason we allow the SMT solver to place the allocated objects. We existentially quantify the starting location for each allocation, such that, if objects can be rearranged to cause an error to occur, they will be.

In our encoding, memory is modeled by three arrays: *heap*, *alloc*, and *objsize*. The first contains a representation of the heap at a given time (execution step):

$$\text{heap} : \text{Time} \rightarrow \text{Address} \rightarrow \text{Value}$$

The *alloc* array is used to track whether a memory address is allocated or not:

$$\text{alloc} : \text{Time} \rightarrow \text{Address} \rightarrow \text{Bool}$$

If some address is not allocated at the time of being accessed, this corresponds to a segmentation fault. The $objsize$ array is used to track the size of allocated objects at a given time and memory address:

$$objsize : \text{Time} \rightarrow \text{Address} \rightarrow \text{Nat} \cup \{\perp\}$$

Note that this array contains \perp (encoded as -1) for memory locations that are allocated, but not at the beginning of an object.

4.2 Encoding to SMT

The procedure **feasible** ats bound checks for the feasibility of counter-examples of bounded length in an ATS. If such a counter-example exists, it returns a mapping $\mu : \mathbb{L} \rightarrow \text{Structures}$ which associates each vertex with a Kripke structure that provides a concrete interpretation for each variable, function symbol, and the heap memory. If no such counter-example exists, **feasible** returns **None**.

In order to pose the bounded model checking problem as a single SMT problem, we make use of quantifiers. We constrain the solver to start at some symbolic set of initial states constructed by *init* and then for some bounded number of steps, unroll the transition relation of the ATS. The function $\text{tr}(ats, t)$ corresponds to the encoding of the transition relation of the ATS *ats* from time $t - 1$ to time t . The top-level check is encoded as

$$\text{init}() \wedge \forall t. 0 < t < \text{bound} \rightarrow \text{tr}(ats, t).$$

Our encoding makes use of semantic functions $\llbracket \cdot \rrbracket$, which take a state and continuations (to work out what to do next in the translation). In what follows, the encoding of commands is denoted by

$$\llbracket \cdot \rrbracket^C : St \rightarrow (St \rightarrow SMT) \rightarrow (St \rightarrow SMT) \rightarrow SMT,$$

which takes a state and two continuations, one for successful transitions and one for transitions to error. The transition relation $\text{tr}(ats, t)$ is encoded as

$$\begin{aligned} \text{tr}(ats, t) = (at(t - 1) = \ell_e \rightarrow at(t) = \ell_e) \wedge \\ \left\{ \bigwedge_{\ell \in \mathbb{L}} at(t - 1) = \ell \rightarrow \left(\bigvee_{(\ell, \ell') \in \mathbb{E}} \llbracket \kappa(\ell, \ell') \rrbracket^C \sigma sk ek \right) \right\}, \end{aligned}$$

where \mathbb{L} and \mathbb{E} are the sets of vertices and edges of the ATS and the function $at(t)$ encodes the control vertex at time t .

We use the 4-tuple $\langle vars, heap, alloc, objsize \rangle$ to represent a state of the system, where *vars* is the set of variables in the ATS. This is used as the source of generating the corresponding time-stamped variables in the encoding. For efficiency reasons, the implementation also keeps flags for if and when the state was last updated. The arrays in the initial environment σ_0 are empty.

The top-level command encoding takes the two continuations, one to signify a successful transition $sk = \lambda\sigma. STEP(t, \ell, \sigma)$ and another for transitions to

the error $ek = \lambda\sigma. \text{ERR}(t, \sigma)$. If the command completes without error, the command threads the modified state to sk , otherwise it threads the modified state to the error continuation. Once the error continuation is followed, the top-level encoding tr ensures that the system will stay in the error state. These continuations allow for a clean representation of the ATS that maximizes the use of if-then-else structures and minimizes general disjunctions. Threading the state also allowed us to reduce the number of quantifiers used in the problem by using if-then-else constructs instead of quantified uninterpreted functions, so long as we are in the same block as previous heap updates.

The initial continuations end with the $STEP$ and ERR predicates which are defined as follows. $STEP(t, \ell, \sigma)$ asserts that $at(t) = \ell$ to ensure the transition of the vertex to the next time step. (A transition to ℓ_e is explicitly disallowed.) Furthermore, it preserves all the values from the current block that must be preserved ($heap$ if modified, $alloc$ and $objsize$ if modified, as well as all variables). Lastly, $STEP(t, \ell, \sigma)$ asserts that $pure(\delta \ell)$, the pure consequences of the Separation Logic assertion at ℓ , hold at time t . The $ERR(t, \sigma)$ predicate is like $STEP(t, \ell, \sigma)$ except that the transition must be to ℓ_e and $pure(\delta \ell) = true$.

We now describe the encoding of commands, concentrating on the memory-related commands **malloc**, **free** and **store**. A forthcoming tech report [6] gives a full definition of the encoding for the other commands.

The **malloc** command produces a new function for the $alloc$ array. It uses a fresh variable to store the location. We cannot simply constrain the target variable x , because it may already have been assigned a value and thus is not unconstrained. By introducing a fresh variable, f , constraining it and updating x to be equal to f , we achieve the desired effect. The seemingly odd constraint that $f \leq f + s$, given that $s \geq 0$ exists because of the modular behavior of arithmetic in the bit-vector theory. Without this constraint, memory would be allowed to wrap around past zero. While this behavior should be prohibited by the constraint from *init* that location 0 is always deallocated, adding this constraint provides performance benefits. Formally, the encoding of **malloc** is defined by

```

$$\llbracket x := \text{malloc}(s) \rrbracket^C \sigma sk ek = \text{let } \langle vars, heap, alloc, objsize \rangle = \sigma \text{ in}$$


$$\quad \text{let } f = \text{gensym}() \text{ in}$$


$$\quad \text{let } z = (\llbracket s \rrbracket^{\text{EXP}} \sigma) \text{ in}$$


$$\quad \forall i. f \leq i < f + z \rightarrow alloc(i) = false \wedge$$


$$\quad \forall i. f \leq i < f + z \rightarrow objsize(i) = -1 \wedge$$


$$\quad \text{let } a' = \lambda a. \text{ite}(f \leq a < f + z, \text{true}, \text{alloc } a) \text{ in}$$


$$\quad \text{let } s' = \lambda a. \text{ite}(f = a, s, \text{objsize } a) \text{ in}$$


$$\quad (sk \langle vars \oplus [x \mapsto f], heap, a', s' \rangle),$$

```

where $\text{gensym}()$ represents the introduction of a fresh symbol.

The **free** command is similar to **malloc**, except that it relies upon the values in the $objsize$ array instead of the $alloc$ array. It requires that the $objsize$ of the freed address have a value other than -1 , whose value indicates no value in the

size array. This value in $objsize$ indicates how many successive entries in $alloc$, starting at address x , need to be set back to *false*. Formally,

$$\llbracket \text{free}(x) \rrbracket^C \sigma sk ek = \text{let } \langle vars, heap, alloc, objsize \rangle = \sigma \text{ in} \\ \quad \text{let } f = \text{gensym}() \text{ in} \\ \quad \text{let } s = (objsize\ x) \text{ in} \\ \quad \text{let } a' = \lambda a. \text{ite}(f \leq a < f + s, \text{false}, alloc\ a) \text{ in} \\ \quad \text{let } s' = \lambda a. \text{ite}(f = a, -1, objsize\ a) \text{ in} \\ \quad \text{let } \sigma' = \langle vars, heap, a', s' \rangle \text{ in} \\ \quad \text{ite}(x = 0, (sk\ \sigma), \text{ite}(s \neq -1, (sk\ \sigma'), (ek\ \sigma))) ,$$

The **store** command first checks the precondition $(alloc\ x)$, which is that the memory at the target address is in fact allocated. If this precondition holds, the execution is allowed to continue with the updated state where $heap$ has been assigned to a new function. Conversely, the execution continues at ℓ_e , assuming that the state was not updated as required by the command. Formally, we have

$$\llbracket *x = y \rrbracket^C \sigma sk ek = \text{let } \langle vars, heap, alloc, objsize \rangle = \sigma \text{ in} \\ \quad \text{let } heap' = \lambda a. \text{ite}(a = x, y, (heap\ a)) \text{ in} \\ \quad \text{ite}((alloc\ x), (sk\ \langle vars, heap', alloc, objsize \rangle), (ek\ \sigma)) .$$

5 Doomed State Synthesis

We define the **diagnose** procedure for identifying doomed states, i.e., for states for which abstraction was too aggressive, and so can be passed to a refinement procedure. Our procedure works as follows: It iterates through the edges of the abstract counter-example, to determine at which of them the widening operator has abstracted too coarsely. It does this by analyzing a new, temporary ATS in which the prefix of the cutpoint ℓ' has been replaced with a program fragment that constructs states which satisfy $\delta\ell'$, the formula at that cutpoint. We then use the **feasible** procedure to search for a concrete counter-example in this new ATS. If a counter-example is found, then it returns the discontinuity (ℓ, ℓ') together with the doomed state obtained by looking up ℓ' in the concrete trace μ . The **diagnose** procedure is depicted in Algorithm 2.

Executing the code generated by **prefix** Q produces states that satisfy Q . Algorithm 3 defines **prefix**, where the generated pseudo-code is shorthand for standard control-flow graph construction, and the **local v in C** form is short for $C[v'/v]; v' = \text{nondet}()$ where v' is fresh.

The model generation assumes a model finder for first-order logic, so first-order formulas F are simply assumed. Existential quantification is synthesized using non-deterministic assignment, reverse engineering Floyd's assignment axiom. Disjunction is translated into a non-deterministic branch, that is, disjunction of commands. Nothing need be done to synthesize **emp** since it is a sub-heap

Algorithm 2. Doomed state search

```

let diagnose <math>\langle \mathbb{L}, \mathbb{E}, \ell_0, \kappa, \delta \rangle</math> bound =
  for <math>(\ell, \ell') \in \mathbb{E} \text{ do}</math>
    if widened <math>(\ell, \ell')</math>
      let <math>\langle \mathbb{L}_n, \mathbb{E}_n, \ell_n, \kappa_n \rangle = \text{cfg\_of } ((\text{prefix } \delta \ell'); \text{ goto } \ell') \text{ in}</math>
      let mod_ats = <math>\langle \mathbb{L} \cup \mathbb{L}_n, \mathbb{E} \cup \mathbb{E}_n, \ell_n, \kappa \cup \kappa_n, \delta \cup (\mathbb{L}_n \times \{\text{emp}\}) \rangle</math> in
      match feasible mod_ats bound with
        | None ->
          continue
        | <math>\mu \rightarrow</math>
          return <math>((\ell, \ell'), \mu(\ell'))</math>
  return None

```

of any heap. Points-to formulas are synthesized by a `malloc()` call, and separating conjunction is mapped to sequential composition. This has the effect of encoding the core partiality in the semantics of `*` into the freshness guarantee and non-determinism provided by allocation, meaning that correctly generating models relies on an accurate treatment of allocation. Lastly, lists are synthesized by using a loop to realize induction on the list length. As an example, `prefix ls(_SLL_ENTRY, k, , p, , q)` is realized by, after slight simplification:

```

local l,f,a;
l = k; f = p;
for ( ; l > 0; --l) {
  a = malloc(sizeof(sll));
  assume(f == a);
  f = f->Flink;
}
assume(f = q & l = 0);

```

Lemma 1. *Every reachable state of `prefix Q` satisfies `Q`.*

Theorem 1. *The `abstraction_refinement` procedure is a sound analysis.*

Proof. The procedure only returns `Safe` when the abstract interpreter in `analyze` did in fact find a proof; this result is correct as long as the `refine` procedure maintains the fact that the abstraction is in fact a valid abstraction. In case the procedure returns `Unsafe`, it has found a concrete counter-example which witnesses the fact that the program is in fact unsafe. In all other cases, the procedure returns `PossiblyUnsafe`, which does not harm the soundness of the analysis. \square

Note that approach for doomed state synthesis has the effect of translating separation logic formulas to code, and then in the feasibility checker, to first-order logic formulas. It would be possible to compose these two translations and translate separation logic formulas to first-order logic directly, but the result would be more difficult to understand, and would impede reuse in the implementation.

Algorithm 3. Prefix synthesis

```

let prefix Q =
  match Q with
  | F ->
    assume(F)
  |  $\exists x. Q$  ->
    local x in (prefix Q)
  |  $Q_0 \vee \dots \vee Q_N$  ->
    if nondet() then (prefix  $Q_0$ )
    else (prefix  $Q_1 \vee \dots \vee Q_N$ )
  | emp ->
    nop
  |  $Q * R$  ->
    (prefix Q); (prefix R)
  |  $l \mapsto [r; o_1 : e_1; \dots; o_N : e_N]$  ->
    local a in
      a = malloc(sizeof(typeof(r)));
      assume(a = l);
      *l.o1 = e1; ...; *l.oN = eN
  | ls( $\Lambda, k, p, f, b, n$ ) ->
    local l, w, x in
      l = k; w = p; x = f;
      for (; l > 0; l = l - 1) {
        local y, z in
          (prefix  $\Lambda(w, x, y, z)$ );
        w = y; x = z
      }
    assume(l = 0  $\wedge$  w = b  $\wedge$  x = n)

```

Note that the separation logic formulas are not precisely expressible in first-order logic due to transitive closure used to interpret the list predicate and second-order quantification implicit in the semantics of the $*$ connective. So a direct translation must under-approximate, and the ways that the transitive closure and second-order quantification interact make this nontrivial. The translation via the model-construction code avoids eagerly constructing formulas of size exponential in the bound, unlike a naive “blasting” approach. Additionally, the translation via code approach potentially allows the solver to unroll loops in the generation of a model of the separation logic formula guided by the path to error, where a direct translation would blindly generate the first order logic formula without any guidance.

6 Experimental Evaluation

There are two motivations in undertaking the work described in this paper. One is to make precise the notion of abstraction failure diagnosis in separation logic

shape analyses. The other, a more practical one, is to use this understanding to improve the quality of results of SLAYER runs. We implemented feasibility checking and diagnosis in SLAYER. This alone has improved SLAYER regression tests, in particular turning around two dozen known unsafe tests from `PossiblyUnsafe` to definitely `Unsafe`.

We also implemented a simple pattern `refine` procedure. SLAYER keeps a set of active and inactive abstraction patterns. When widening admits a doomed state s , this diagnosis is fed into SLAYER’s shape discovery module in order to select a pattern to eliminate the doomed state. The basic algorithm for refinement is to enumerate the inactive patterns, for each one widen to s' using the active patterns plus the chosen one, and then check if s entails s' . If not, keep the chosen pattern active; otherwise it keeps looking. This is a simple automatic refinement procedure, and we can imagine more sophisticated schemes. For instance, to deal with more complex programs, we could try with all the inactive patterns and then minimize akin to unsat core minimization in MaxSAT.

Table I presents some experimental results. The programs are taken from the SLAYER test suite, and so are biased towards control (rather than data), traversal through linked lists, pointer arithmetic, etc. The table gives the results for SLAYER without and with this simple pattern refinement scheme. The second column indicates that these are all tests where SLAYER previously reported an inconclusive result, in the time indicated in the third column. The fourth column reports the result using the techniques described here, either `Unsafe` indicating a concrete counter-example of memory safety was found, or `Safe` indicating that a memory safety proof was found after abstraction refinement, or `PossUnsafe` indicating a result that remains inconclusive. The fifth column reports the additional time taken either for feasibility checking or for diagnosis and refinement, indicated as the sum of shape analysis and feasibility checking times.

7 Related Work

Counter-Example Guided Abstraction Refinement (CEGAR) inspired this work. SLAYER’s implementation attempts to mirror the primary steps of the algorithm without requiring weakest precondition or general conjunction. There have been many implementations of CEGAR, though it is most popularly used with predicate abstraction as in the SLAM tool [12]. Other implementations include one by Clarke et al [13] applied to hardware, the BLAST project [23], MAGIC [11] and SATABS [15]. Obtaining the initial abstraction is not addressed by CEGAR, but there are several techniques, including existential abstraction [14] and predicate abstraction [16, 22].

Our feasibility checking algorithm is an implementation of bounded model checking [7] and is most closely related to the CBMC [12] bounded model checker for C programs. We implement bounded model checking as a single large problem and leave the task of determining unrolling to the SMT solver. This differs from CBMC in that CBMC does explicit unrolling.

Instead of bounding the depth of the search, it is possible to bound the breadth of the search by using a symbolic or concolic testing technique. Tools like EXE [9],

Table 1. SLAYER versus SLAYER + Feasibility Checking and Pattern Refinement

Test	SLAYER		SLAYER + Diagnosis	
	Result	Time	Disproved/Refined Result	Time
T2_n_19	PossUnsafe	0.031	Unsafe	+0.078
T2_n_1b	PossUnsafe	0.016	Unsafe	+0.062
T2_n_34	PossUnsafe	0.031	Unsafe	+0.140
T2_n_38	PossUnsafe	0.078	Unsafe	+0.421
T2_p_38	PossUnsafe	0.515	Unsafe	+12.230
T2_p_50	PossUnsafe	0.062	Unsafe	+0.562
T2_p_62	PossUnsafe	0.078	Unsafe	+0.546
changing_truth_value	PossUnsafe	0.062	Unsafe	+1.373
complicated_safe	PossUnsafe	0.140	PossUnsafe	+89.279
complicated_unsafe	PossUnsafe	0.156	Unsafe	+2.309
no_loops_unsafe	PossUnsafe	0.016	Unsafe	+0.140
simple_loop_unsafe	PossUnsafe	0.109	Unsafe	+0.078
very_simple_unsafe	PossUnsafe	0.000	Unsafe	+0.016
csl_remove_unsafe	PossUnsafe	0.499	Unsafe	+1.342
cleanup_isochresourcedata	PossUnsafe	0.796	Unsafe	+23.306
array_in_formal	PossUnsafe	0.016	Unsafe	+0.094
deref_NULL	PossUnsafe	0.016	Unsafe	+0.031
free_free	PossUnsafe	0.000	Unsafe	+0.016
free_local	PossUnsafe	0.000	Unsafe	+0.047
if_pointer	PossUnsafe	0.000	Unsafe	+0.016
sized_arrays	PossUnsafe	0.016	Unsafe	+0.078
store_to_0	PossUnsafe	0.000	Unsafe	+0.031
sll_copy_unsafe	PossUnsafe	0.218	Unsafe	+2.293
list_of_objects	PossUnsafe	0.140	Safe	+0.230+5.975

KLEE [8], DART [20], CUTE [32] and SAGE [21] are well tuned to rapidly search large code bases looking for memory violations, assertion violations and arithmetic bugs. They could benefit from the reduction in state-space that searching an abstract transition system provides, but we preferred the guarantee that all paths were searched up to a specific depth and thus did not use these techniques.

Instead of bounding the search space using an abstract transition system, a symbolic testing engine could use a heuristic that guides it to reach certain program points. Ma, et al [28] explore this approach but do not attempt to use it to guide another analysis.

Our approach to refinement is complementary to the pattern discovery and synthesis such as [3], but refinement is not the only way to improve the widening. By adding more information to the abstraction, such as numerics [29,30], the proofs will become more likely to succeed. This does not preclude a refinement phase, however. This would reduce the number of times refinement was needed, but widening still loses data values and thus might lose information such as sortedness of a fixed length list that our counter-example generation would know.

An abstraction refinement technique applicable to shape analysis has been proposed [27]. This technique refines the generalization of predicate abstraction [31] used by TVLA [26]. Rather than being guided by counter-examples, refinement is directed by either the syntax of an asserted formula with an indeterminate valuation, or by detection of precision loss by the abstraction during the proof attempt, without an indication that the lost precision is relevant to the proof failure.

A problem similar to feasibility checking has been investigated in the context of TVLA [19]. There, a bounded breadth-first search through program paths and a bounded model finder for first-order logic is used, in contrast to our single search encoded into SMT. It seems likely that this feasibility checker could be combined with our diagnosis technique to also obtain an abstraction failure diagnosis for shape analyses based on 3-valued logic.

8 Conclusion

We have presented a method for diagnosing abstraction failure in separation logic-based analyses. To do this, we use a new algorithm to pinpoint where abstraction failed based on a concrete counter-example. We generate this concrete counter-example with a bounded model checker that precisely analyzes abstract transition systems. These techniques have been implemented and evaluated using a pattern-based abstraction refinement scheme in SLAYER, a tool for automated analysis of low-level C programs, and have become an invaluable aid in debugging failed SLAYER runs and refining the definition of abstraction. With this contribution, we look forward to finding new automatic refinement algorithms that significantly improve the capacity and precision of shape analyses.

References

1. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. *Commun. ACM* 54(7) (2011)
2. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
3. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O’Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
4. Berdine, J., Calcagno, C., O’Hearn, P.W.: Symbolic Execution with Separation Logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
5. Berdine, J., Cook, B., Ishtiaq, S.: SLAYER: Memory Safety for Systems-Level Code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
6. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.: Diagnosing abstraction failure for separation logic-based analyses. Tech. Rep. MSR-TR-2012-44, Microsoft Research, Cambridge (April 2012)

7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
8. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)
9. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. In: CCS (2006)
10. Calcagno, C., Yang, H., O’Hearn, P.W.: Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 108–119. Springer, Heidelberg (2001)
11. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Software Eng. 30(6) (2004)
12. Clarke, E., Kroning, 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)
13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
14. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16 (1994)
15. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
16. Colón, M., Uribe, T.E.: Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 293–304. Springer, Heidelberg (1998)
17. 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)
18. Elkarakibieh, B., Godefroid, P., Levin, M.Y.: Precise pointer reasoning for dynamic test generation. In: ISSTA (2009)
19. Erez, G.: Generating concrete counterexamples for sound abstract interpretation. Master’s thesis, School of Computer Science, Tel-Aviv University, Israel (2004)
20. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI (2005)
21. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
22. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
23. 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)
24. Ishtiaq, S., O’Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL (2001)
25. Jussila, T., Biere, A.: Compressing BMC encodings with QBF. Electr. Notes Theor. Comput. Sci. 174(3), 45–56 (2007)
26. Lev-Ami, T., Sagiv, M.: TVLA: A System for Implementing Static Analyses. In: SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)

27. Loginov, A., Reps, T., Sagiv, M.: Abstraction Refinement via Inductive Learning. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 519–533. Springer, Heidelberg (2005)
28. Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed Symbolic Execution. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 95–111. Springer, Heidelberg (2011)
29. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic Strengthening for Shape Analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)
30. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL (2010)
31. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
32. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. SIGSOFT Softw. Eng. Notes 30 (2005)
33. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. In: FMCAD (2010)

A Method for Symbolic Computation of Abstract Operations^{*}

Aditya Thakur¹ and Thomas Reps^{1,2}

¹ University of Wisconsin, Madison, WI, USA

² GrammaTech, Inc.; Ithaca, NY, USA

Abstract. This paper helps to bridge the gap between (i) the use of logic for specifying program semantics and performing program analysis, and (ii) abstract interpretation. Many operations needed by an abstract interpreter can be reduced to the problem of *symbolic abstraction*: the symbolic abstraction of a formula φ in logic \mathcal{L} , denoted by $\widehat{\alpha}(\varphi)$, is the most-precise value in abstract domain \mathcal{A} that over-approximates the meaning of φ . We present a parametric framework that, given \mathcal{L} and \mathcal{A} , implements $\widehat{\alpha}$. The algorithm computes successively better over-approximations of $\widehat{\alpha}(\varphi)$. Because it approaches $\widehat{\alpha}(\varphi)$ from “above”, if it is taking too much time, a safe answer can be returned at any stage.

Moreover, the framework is “dual-use”: in addition to its applications in abstract interpretation, it provides a new way for an SMT (Satisfiability Modulo Theories) solver to perform unsatisfiability checking: given $\varphi \in \mathcal{L}$, the condition $\widehat{\alpha}(\varphi) = \perp$ implies that φ is unsatisfiable.

1 Introduction

This paper concerns the connection between abstract interpretation and logic. Like several previous papers [29,37,21,12], our work is based on the insight that many of the key operations needed by an abstract interpreter can be reduced to the problem of *symbolic abstraction* [29].

Suppose that \mathcal{A} is an abstract domain with concretization function $\gamma : \mathcal{A} \rightarrow \mathcal{C}$. Given a formula φ in logic \mathcal{L} , let $\llbracket \varphi \rrbracket$ denote the meaning of φ —i.e., the set of concrete states that satisfy φ . The *symbolic abstraction* of φ , denoted by $\widehat{\alpha}(\varphi)$, is the best \mathcal{A} value that over-approximates $\llbracket \varphi \rrbracket$: $\widehat{\alpha}(\varphi)$ is the unique value $A \in \mathcal{A}$ such that (i) $\llbracket \varphi \rrbracket \subseteq \gamma(A)$, and (ii) for all $A' \in \mathcal{A}$ for which $\llbracket \varphi \rrbracket \subseteq \gamma(A')$, $A \sqsubseteq A'$.

This paper presents a new framework for performing symbolic abstraction, discusses its properties, and presents several instantiations for various logics and

* This research is supported, in part, by NSF under grants CCF-{0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 10-M-0251}, by ARL under grant W911NF-09-1-0413, by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088; and by DARPA under cooperative agreement HR0011-12-2-0012. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

abstract domains. In addition to providing insight on fundamental limits, the new algorithm for $\widehat{\alpha}$ also performs well: our experiments show that it is 11.3 times faster than a competing method [29][21][12], while finding dataflow facts (i.e., invariants) that are equally precise at 76.9% of a program’s basic blocks, better (tighter) at 19.8% of the blocks, and worse (looser) at only 3.3% of the blocks.

Most-Precise Abstract Interpretation. Suppose that $\mathcal{G} = \mathcal{C} \xleftarrow[\alpha]{\gamma} \mathcal{A}$ is a Galois connection between concrete domain \mathcal{C} and abstract domain \mathcal{A} . Then the “best transformer” [7], or best abstract post operator for transition τ , denoted by $\widehat{\text{Post}}[\tau] : \mathcal{A} \rightarrow \mathcal{A}$, is the most-precise abstract operator possible, given \mathcal{A} , for the concrete post operator for τ , $\text{Post}[\tau] : \mathcal{C} \rightarrow \mathcal{C}$. $\widehat{\text{Post}}[\tau]$ can be expressed in terms of α , γ , and $\text{Post}[\tau]$, as follows [7]: $\widehat{\text{Post}}[\tau] = \alpha \circ \text{Post}[\tau] \circ \gamma$. This equation defines the limit of precision obtainable using abstraction \mathcal{A} . However, it is non-constructive; it does not provide an *algorithm*, either for applying $\widehat{\text{Post}}[\tau]$ or for finding a representation of the function $\widehat{\text{Post}}[\tau]$. In particular, in many cases, the application of γ to an abstract value would yield an intermediate result—a set of concrete states—that is either infinite or too large to fit in computer memory.

Symbolic Abstract Operations. The aforementioned problem with applying γ can be side-stepped by working with *symbolic* representations of sets of states (i.e., using formulas in some logic \mathcal{L}). The use of \mathcal{L} formulas to represent sets of states is convenient because logic can also be used for specifying a language’s concrete semantics; i.e., the concrete semantics of a transformer $\text{Post}[\tau]$ can be stated as a formula $\varphi_\tau \in \mathcal{L}$ that specifies the relation between input states and output states. However, the symbolic approach introduces a new challenge: how to bridge the gap between \mathcal{L} and \mathcal{A} [29]. In particular, we need to develop (i) algorithms to handle interconversion between formulas of \mathcal{L} and abstract values in \mathcal{A} , and (ii) symbolic versions of the operations that form the core repertoire at the heart of an abstract interpreter.

1. $\widehat{\gamma}(A)$: Given an abstract value $A \in \mathcal{A}$, the *symbolic concretization* of A , denoted by $\widehat{\gamma}(A)$, maps A to a formula $\widehat{\gamma}(A)$ such that A and $\widehat{\gamma}(A)$ represent the same set of concrete states (i.e., $\gamma(A) = [\![\widehat{\gamma}(A)]\!]$).
2. $\widehat{\alpha}(\varphi)$: Given $\varphi \in \mathcal{L}$, the symbolic abstraction of φ , denoted by $\widehat{\alpha}(\varphi)$, maps φ to the best value in \mathcal{A} that over-approximates $[\![\varphi]\!]$ (i.e., $\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!])$).
3. $\widehat{\text{Assume}}[\varphi](A)$: Given $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$, $\widehat{\text{Assume}}[\varphi](A)$ returns the best value in \mathcal{A} that over-approximates the meaning of φ in concrete states described by A . That is, $\widehat{\text{Assume}}[\varphi](A)$ equals $\alpha([\![\varphi]\!] \cap \gamma(A))$.
4. Creation of a representation of $\widehat{\text{Post}}[\tau]$: Some intraprocedural [15] and many interprocedural [32][22] dataflow-analysis algorithms operate on instances of an abstract datatype \mathcal{T} that (i) represents a family of abstract functions (or relations), and (ii) is closed under composition and join. By “creation of a representation of $\widehat{\text{Post}}[\tau]$ ”, we mean finding the best instance in \mathcal{T} that over-approximates $\widehat{\text{Post}}[\tau]$.

Several other symbolic abstract operations are discussed in §6

Experience shows that, for most abstract domains, it is easy to write a $\widehat{\alpha}$ function (item ① [29]). The other three operations are inter-related. $\widehat{\alpha}$ (item ②) can be reduced to Assume (item ③) as follows: $\widehat{\alpha}(\varphi) = \widehat{\text{Assume}}[\varphi](\top)$. Item ④ can be reduced to item ② as follows: The concrete post operator $\text{Post}[\tau]$ corresponds to a formula $\varphi_\tau \in \mathcal{L}$ that expresses the transition relation between input states and output states. An instance of abstract datatype \mathcal{T} in item ④ represents an abstract-domain element that denotes an over-approximation of $[\varphi_\tau]$. $\widehat{\alpha}(\varphi_\tau)$ computes the best instance in \mathcal{T} that over-approximates $[\varphi_\tau]$.

This paper presents a parametric framework that, for some abstract domains, is capable of performing most-precise abstract operations in the limit. Because the method approaches its result from “above”, if the computation takes too much time, it can be stopped to yield a safe result—i.e., an over-approximation to the best abstract operation—at any stage. Thus, the framework provides a tunable algorithm that offers a performance-versus-precision trade-off. We replace “ $\widehat{}$ ” with “ $\widetilde{}$ ” to denote over-approximating operators—e.g., $\widetilde{\alpha}(\varphi)$, $\widetilde{\text{Assume}}[\varphi](A)$, and $\widetilde{\text{Post}}[\tau](A)$.¹

Key Insight. In [35], we showed how Stålmarck’s method [33], an algorithm for satisfiability checking of propositional formulas, can be explained using abstract-interpretation terminology—in particular, as an instantiation of a more general algorithm, Stålmarck[\mathcal{A}], that is parameterized by a (Boolean) abstract domain \mathcal{A} and operations on \mathcal{A} . The algorithm that goes by the name “Stålmarck’s method” is one instantiation of Stålmarck[\mathcal{A}] with a certain abstract domain.

Abstract value A' is a *semantic reduction* [7] of A with respect to φ if (i) $\gamma(A') \cap [\varphi] = \gamma(A) \cap [\varphi]$, and (ii) $A' \sqsubseteq A$. At each step, Stålmarck[\mathcal{A}] holds some $A \in \mathcal{A}$; each of the so-called “propagation rules” employed in Stålmarck’s method improves A by finding a semantic reduction of A with respect to φ .

The key insight of the present paper is that there is a connection between Stålmarck[\mathcal{A}] and $\widetilde{\alpha}_{\mathcal{A}}$. In essence, to check whether a formula φ is unsatisfiable, Stålmarck[\mathcal{A}] computes $\widetilde{\alpha}_{\mathcal{A}}(\varphi)$ and performs the test “ $\widetilde{\alpha}_{\mathcal{A}}(\varphi) = \perp_{\mathcal{A}}$ ”? If the test succeeds, it establishes that $[\varphi] \subseteq \gamma(\perp_{\mathcal{A}}) = \emptyset$, and hence that φ is unsatisfiable.

In this paper, we present a generalization of Stålmarck’s algorithm to *richer logics*, such as quantifier-free linear rational arithmetic (QF_LRA) and quantifier-free bit-vector arithmetic (QF_BV). Instead of only using a Boolean abstract domain, the generalized method of this paper also uses *richer abstract domains*, such as the polyhedral domain [8] and the bit-vector affine-relations domain [12]. By this means, we obtain algorithms for computing $\widetilde{\alpha}$ for these richer abstract domains. The bottom line is that our algorithm is “dual-use”: (i) it can be used by an abstract interpreter to compute $\widetilde{\alpha}$ (and perform other symbolic abstract operations), and (ii) it can be used in an SMT (Satisfiability Modulo Theories) solver to determine whether a formula is satisfiable.

¹ $\widetilde{\text{Post}}[\tau]$ is used by Graf and Saïdi [14] to mean a different state transformer from the one that $\widetilde{\text{Post}}[\tau]$ denotes in this paper. Throughout the paper, we use $\widetilde{\text{Post}}[\tau]$ solely to mean an over-approximation of $\widehat{\text{Post}}[\tau]$; thus, our notation is not ambiguous.

Because we are working with more expressive logics, our algorithm uses several ideas that go beyond what is used in either Stålmarck’s method [33] or in Stålmarck[\mathcal{A}] [35]. The methods described in this paper are also quite different from the huge amount of recent work that uses decision procedures in program analysis. It has become standard to reduce program paths to formulas by encoding a program’s actions in logic (e.g., by symbolic execution) and calling a decision procedure to determine whether a given path through the program is feasible. In contrast, the techniques described in this paper adopt—and adapt—the key ideas from Stålmarck’s method to create new algorithms for key program-analysis operations. Finally, the methods described in this paper are quite different from previous methods for symbolic abstraction [29,37,21,12], which all make repeated calls to an SMT solver.

Contributions. The contributions of the paper can be summarized as follows:

- We present a connection between symbolic abstraction and Stålmarck’s method for checking satisfiability (§2).
- We present a generalization of Stålmarck’s method that lifts the algorithm from propositional logic to richer logics (§3).
- We present a new parametric framework that, for some abstract domains, is capable of performing most-precise abstract operations in the limit, including $\widehat{\alpha}(\varphi)$ and $\widehat{\text{Assume}}[\varphi](A)$, as well as creating a representation of $\widehat{\text{Post}}[\mathcal{T}]$. Because the method approaches most-precise values from “above”, if the computation takes too much time it can be stopped to yield a sound result.
- We present instantiations of our framework for two logic/abstract-domain pairs: QF_BV/KS and QF_LRA/Polyhedra, and discuss completeness (§4).
- We present experimental results that illustrate the dual-use nature of our framework. One experiment uses it to compute abstract transformers, which are then used to generate invariants; another experiment uses it for checking satisfiability (§5).

§6 discusses other symbolic operations. §7 discusses related work. Proofs can be found in [36].

2 Overview

We now illustrate the key points of our Stålmarck-inspired technique using two examples. The first shows how our technique applies to computing abstract transformers; the second describes its application to checking unsatisfiability.

The top-level, overall goal of Stålmarck’s method can be understood in terms of the operation $\widetilde{\alpha}(\psi)$. However, Stålmarck’s method is recursive (counting down on a parameter k), and the operation performed at each recursive level is the slightly more general operation $\widetilde{\text{Assume}}[\psi](A)$. Thus, we will discuss $\widetilde{\text{Assume}}$.

Example 1. Consider the following x86 assembly code

```
L1: cmp eax, 2    L2: jz L4    L3: ...
```

The instruction at L1 sets the zero flag (`zf`) to true if the value of register `eax` equals 2. At instruction L2, if `zf` is true the program jumps to location L4 (not

seen in the code snippet) by updating the value of the program counter (pc) to L4; otherwise, control falls through to program location L3. The transition formula that expresses the state transformation from the beginning of L1 to the beginning of L4 is thus $\varphi = (\text{zf} \Leftrightarrow (\text{eax} = 2)) \wedge (\text{pc}' = \text{ITE}(\text{zf}, \text{L4}, \text{L3})) \wedge (\text{pc}' = \text{L4}) \wedge (\text{eax}' = \text{eax})$. (φ is a QF_BV formula.)

Let \mathcal{A} be the abstract domain of affine relations over the x86 registers. Let $A_0 = \top_{\mathcal{A}}$, the empty set of affine constraints over input-state and output-state variables. We now describe how our algorithm creates a representation of the \mathcal{A} transformer for φ by computing $\widetilde{\text{Assume}}[\varphi](A_0)$. The result represents a sound abstract transformer for use in affine-relation analysis (ARA) [27, 21, 12]. First, the ITE term in φ is rewritten as $(\text{zf} \Rightarrow (\text{pc}' = \text{L4})) \wedge (\neg \text{zf} \Rightarrow (\text{pc}' = \text{L3}))$. Thus, the transition formula becomes $\varphi = (\text{zf} \Leftrightarrow (\text{eax} = 2)) \wedge (\text{zf} \Rightarrow (\text{pc}' = \text{L4})) \wedge (\neg \text{zf} \Rightarrow (\text{pc}' = \text{L3})) \wedge (\text{pc}' = \text{L4}) \wedge (\text{eax}' = \text{eax})$.

Next, *propagation rules* are used to compute a semantic reduction with respect to φ , starting from A_0 . The main feature of the propagation rules is that they are “local”; that is, they make use of only a small part of formula φ to compute the semantic reduction.

1. Because φ has to be true, we can conclude that each of the conjuncts of φ are also true; that is, $\text{zf} \Leftrightarrow (\text{eax} = 2)$, $\text{zf} \Rightarrow (\text{pc}' = \text{L4})$, $\neg \text{zf} \Rightarrow (\text{pc}' = \text{L3})$, $\text{pc}' = \text{L4}$, and $\text{eax}' = \text{eax}$ are all true.
2. Suppose that we have a function $\mu\tilde{\alpha}_{\mathcal{A}}$ such that for a literal $l \in \mathcal{L}$, $A' = \mu\tilde{\alpha}_{\mathcal{A}}(l)$ is a sound overapproximation of $\widehat{\alpha}(l)$. Because the literal $\text{pc}' = \text{L4}$ is true, we conclude that $A' = \mu\tilde{\alpha}_{\mathcal{A}}(\text{pc}' = \text{L4}) = \{\text{pc}' - \text{L4} = 0\}$ holds, and thus $A_1 = A_0 \sqcap A' = \{\text{pc}' - \text{L4} = 0\}$, which is a semantic reduction of A_0 .
3. Similarly, because the literal $\text{eax}' = \text{eax}$ is true, we obtain $A_2 = A_1 \sqcap \mu\tilde{\alpha}_{\mathcal{A}}(\text{eax}' = \text{eax}) = \{\text{pc}' - \text{L4} = 0, \text{eax}' - \text{eax} = 0\}$.
4. We know that $\neg \text{zf} \Rightarrow (\text{pc}' = \text{L3})$. Furthermore, $\mu\tilde{\alpha}_{\mathcal{A}}(\text{pc}' = \text{L3}) = \{\text{pc}' - \text{L3} = 0\}$. Now $\{\text{pc}' - \text{L3} = 0\} \sqcap A_2$ is \perp , which implies that $\llbracket \text{pc}' = \text{L3} \rrbracket \cap \gamma(\{\text{pc}' - \text{L4} = 0, \text{eax}' - \text{eax} = 0\}) = \emptyset$. Thus, we can conclude that $\neg \text{zf}$ is false, and hence that zf is true. This value of zf , along with the fact that $\text{zf} \Leftrightarrow (\text{eax} = 2)$ is true, enables us to determine that $A'' = \mu\tilde{\alpha}_{\mathcal{A}}(\text{eax} = 2) = \{\text{eax} - 2 = 0\}$ must hold. Thus, our final semantic-reduction step produces $A_3 = A_2 \sqcap A'' = \{\text{pc}' - \text{L4} = 0, \text{eax}' - \text{eax} = 0, \text{eax} - 2 = 0\}$.

Abstract value A_3 is a set of affine constraints over the registers at L1 (input-state variables) and those at L4 (output-state variables), and can be used for affine-relation analysis using standard techniques (e.g., see [19] or [12, §5]). \square

The above example illustrates how our technique propagates truth values to various subformulas of φ . The process of repeatedly applying propagation rules to compute $\widetilde{\text{Assume}}$ is called 0-assume. The next example illustrates the *Dilemma Rule*, a more powerful rule for computing semantic reductions.

Example 2. Let \mathcal{L} be QF_LRA, and let \mathcal{A} be the polyhedral abstract domain [8]. Consider the formula $\psi = (a_0 < b_0) \wedge (a_0 < c_0) \wedge (b_0 < a_1 \vee c_0 < a_1) \wedge (a_1 < b_1) \wedge (a_1 < c_1) \wedge (b_1 < a_2 \vee c_2 < a_2) \wedge (a_2 < a_0) \in \mathcal{L}$ (see Fig. 1(a)). Suppose that we want to compute $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$.

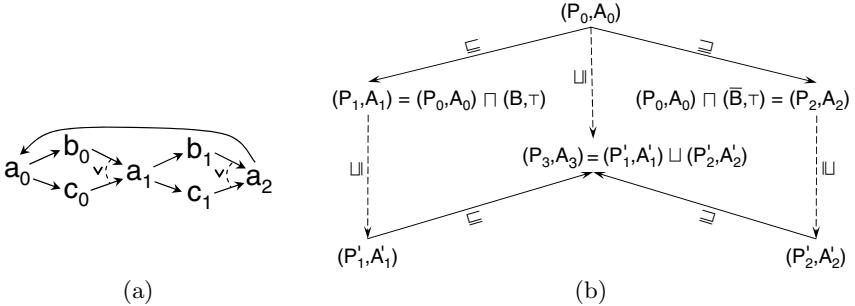


Fig. 1. (a) Inconsistent inequalities in the (unsatisfiable) formula used in Ex. 2. (b) Application of the Dilemma Rule to abstract value (P_0, A_0) . The dashed arrows from (P_i, A_i) to (P'_i, A'_i) indicate that (P'_i, A'_i) is a semantic reduction of (P_i, A_i) .

To make the communication between the truth values of subformulas and the abstract value explicit, we associate a fresh Boolean variable with each subformula of ψ to give a set of *integrity constraints* \mathcal{I} . In this case, $\mathcal{I}_\psi = \{u_1 \Leftrightarrow \bigwedge_{i=2}^8 u_i, u_2 \Leftrightarrow (a_0 < b_0), u_3 \Leftrightarrow (a_0 < c_0), u_4 \Leftrightarrow (u_9 \vee u_{10}), u_5 \Leftrightarrow (a_1 < b_1), u_6 \Leftrightarrow (a_1 < c_1), u_7 \Leftrightarrow (u_{11} \vee u_{12}), u_8 \Leftrightarrow (a_2 < a_0), u_9 \Leftrightarrow (b_0 < a_1), u_{10} \Leftrightarrow (c_0 < a_1), u_{11} \Leftrightarrow (b_1 < a_2), u_{12} \Leftrightarrow (c_1 < a_2)\}$. The integrity constraints encode the structure of ψ via the set of Boolean variables $\mathcal{U} = \{u_1, u_2, \dots, u_{12}\}$. When \mathcal{I} is used as a formula, it denotes the conjunction of the individual integrity constraints.

We now introduce an abstraction over \mathcal{U} ; in particular, we use the Cartesian domain $\mathcal{P} = (\mathcal{U} \rightarrow \{0, 1, *\})_\perp$ in which $*$ denotes “unknown”, and each element in \mathcal{P} represents a set of assignments in $\mathbb{P}(\mathcal{U} \rightarrow \{0, 1\})$. We denote an element of the Cartesian domain as a mapping, e.g., $[u_1 \mapsto 0, u_2 \mapsto 1, u_3 \mapsto *]$, or $[0, 1, *]$ if u_1 , u_2 , and u_3 are understood. $\top_{\mathcal{P}}$ is the element $\lambda u. *$. The “single-point” partial assignment in which variable v is set to b is denoted by $\top_{\mathcal{P}}[v \mapsto b]$.

The variable $u_1 \in \mathcal{U}$ represents the root of ψ ; consequently, the single-point partial assignment $\top_{\mathcal{P}}[u_1 \mapsto 1]$ corresponds to the assertion that ψ is satisfiable. In fact, the models of ψ are closely related to the concrete values in $[\mathcal{I}] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1])$. For every concrete value in $[\mathcal{I}] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1])$, its projection onto $\{a_i, b_i, c_i \mid 0 \leq i \leq 1\} \cup \{a_2\}$ gives us a model of ψ ; that is, $[\psi] = ([\mathcal{I}] \cap \gamma(\top_{\mathcal{P}}[u_1 \mapsto 1]))|_{\{a_i, b_i, c_i \mid 0 \leq i \leq 1\} \cup \{a_2\}}$. By this means, the problem of computing $\widetilde{\text{Assume}}[\psi](\mathcal{T}_{\mathcal{A}})$ is reduced to that of computing $\widetilde{\text{Assume}}[\mathcal{I}]((\top_{\mathcal{P}}[u_1 \mapsto 1], \mathcal{T}_{\mathcal{A}}))$, where $(\top_{\mathcal{P}}[u_1 \mapsto 1], \mathcal{T}_{\mathcal{A}})$ is an element of the reduced product of \mathcal{P} and \mathcal{A} .

Because u_1 is true in $\top_{\mathcal{P}}[u_1 \mapsto 1]$, the integrity constraint $u_1 \Leftrightarrow \bigwedge_{i=2}^8 u_i$ implies that $u_2 \dots u_8$ are also true, which refines $\top_{\mathcal{P}}[u_1 \mapsto 1]$ to $P_0 = [1, 1, 1, 1, 1, 1, 1, *, *, *, *]$. Because u_2 is true and $u_2 \Leftrightarrow (a_0 < b_0) \in \mathcal{I}$, $\mathcal{T}_{\mathcal{A}}$ can be refined using $\mu\widetilde{\alpha}_{\mathcal{A}}(a_0 < b_0) = \{a_0 - b_0 < 0\}$. Doing the same for u_3, u_5, u_6 , and u_8 , refines $\mathcal{T}_{\mathcal{A}}$ to $A_0 = \{a_0 - b_0 < 0, a_0 - c_0 < 0, a_1 - b_1 < 0, a_1 - c_1 < 0, a_2 - a_0 < 0\}$. These steps refine $(\top_{\mathcal{P}}[u_1 \mapsto 1], \mathcal{T}_{\mathcal{A}})$ to (P_0, A_0) via 0-assume.

To increase precision, we need to use the Dilemma Rule, a branch-and-merge rule, in which the current abstract state is split into two (disjoint) abstract states, 0-assume is applied to both abstract values, and the resulting abstract

values are merged by performing a join. The steps of the Dilemma Rule are shown schematically in Fig. 10(b) and described below.

In our example, the value of u_9 is unknown in P_0 . Let $B \in \mathcal{P}$ be $\top_{\mathcal{P}}[u_9 \mapsto 0]$; then \overline{B} , the abstract complement of B , is $\top_{\mathcal{P}}[u_9 \mapsto 1]$. Note that $\gamma(B) \cap \gamma(\overline{B}) = \emptyset$, and $\gamma(B) \cup \gamma(\overline{B}) = \gamma(\top)$. The current abstract value (P_0, A_0) is split into

$$(P_1, A_1) = (P_0, A_0) \sqcap (B, \top) \quad \text{and} \quad (P_2, A_2) = (P_0, A_0) \sqcap (\overline{B}, \top).$$

Now consider 0-assume on (P_1, A_1) . Because u_9 is false, and u_4 is true, we can conclude that u_{10} has to be true, using the integrity constraint $u_4 \Leftrightarrow (u_9 \vee u_{10})$. Because u_{10} holds and $u_{10} \Leftrightarrow (c_0 < a_1) \in \mathcal{I}$, A_1 can be refined with the constraint $c_0 - a_1 < 0$. Because $a_0 - c_0 < 0 \in A_1$, $a_0 - a_1 < 0$ can be inferred. Similarly, when performing 0-assume on (P_2, A_2) , $a_0 - a_1 < 0$ is inferred. Call the abstract values computed by 0-assume (P'_1, A'_1) and (P'_2, A'_2) , respectively. At this point, the join of (P'_1, A'_1) and (P'_2, A'_2) is taken. Because $a_0 - a_1 < 0$ is present in both branches, it is retained in the join. The resulting abstract value is $(P_3, A_3) = ([1, 1, 1, 1, 1, 1, 1, *, *, *, *], \{a_0 - b_0 < 0, a_0 - c_0 < 0, a_1 - b_1 < 0, a_1 - c_1 < 0, a_2 - a_0 < 0, a_0 - a_1 < 0\})$. Note that although P_3 equals P_0 , A_3 is strictly more precise than A_0 (i.e., $A_3 \sqsubset A_0$), and hence (P_3, A_3) is a semantic reduction of (P_0, A_0) with respect to ψ .

Now suppose (P_3, A_3) is split using u_{11} . Using reasoning similar to that performed above, $a_1 - a_2 < 0$ is inferred on both branches, and hence so is $a_0 - a_2 < 0$. However, $a_0 - a_2 < 0$ contradicts $a_2 - a_0 < 0$; consequently, the abstract value reduces to $(\perp_{\mathcal{P}}, \perp_{\mathcal{A}})$ on both branches. Thus, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}}) = \perp_{\mathcal{A}}$, and hence ψ is unsatisfiable. In this way, $\widetilde{\text{Assume}}$ instantiated with the polyhedral domain can be used to decide the satisfiability of a QF-LRA formula. \square

The process of repeatedly applying the Dilemma Rule is called 1-assume. That is, repeatedly some variable $u \in \mathcal{U}$ is selected whose truth value is unknown, the current abstract value is split using $B = \top_{\mathcal{P}}[u \mapsto 0]$ and $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$, 0-assume is applied to each of these values, and the resulting abstract values are merged via join (Fig. 10(b)). Different policies for selecting the next variable on which to split can affect how quickly an answer is found; however, any fair selection policy will return the same answer. The efficacy of the Dilemma Rule is partially due to case-splitting; however, the real power of the Dilemma Rule is due to the fact that it *preserves information learned in both branches when a case-split is “abandoned” at a join point*.

The generalization of the 1-assume algorithm is called k-assume: repeatedly some variable $u \in \mathcal{U}$ is selected whose truth value is unknown, the current abstract value is split using $B = \top_{\mathcal{P}}[u \mapsto 0]$ and $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$; (k-1)-assume is applied to each of these values; and the resulting values are merged via join. However, there is a trade-off: higher values of k give greater precision, but are also computationally more expensive.

For certain abstract domains and logics, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ is complete—i.e., with a high-enough value of k for k-assume, $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ always computes the most-precise \mathcal{A} value possible for ψ . However, our experiments show that $\widetilde{\text{Assume}}[\psi](\top_{\mathcal{A}})$ has very good precision with $k = 1$ (see §5)—which jibes with the observation that, in practice, with Stålmarck’s method for propositional

Algorithm 1. $\widetilde{\text{Assume}}[\varphi](A)$

```

1  $\langle \mathcal{I}, u_\varphi \rangle \leftarrow \text{integrity}(\varphi)$ 
2  $P \leftarrow \top_{\mathcal{P}}[u_\varphi \mapsto 1]$ 
3  $(\tilde{P}, \tilde{A}) \leftarrow \text{k-assume}[\mathcal{I}]((P, A))$ 
4 return  $\tilde{A}$ 

```

Algorithm 2. $0\text{-assume}[\mathcal{I}]((P, A))$

```

1 repeat
2    $(P', A') \leftarrow (P, A)$ 
3   foreach  $J \in \mathcal{I}$  do
4     if  $J$  has the form  $u \Leftrightarrow \ell$  then
5        $(P, A) \leftarrow \text{LeafRule}(J, (P, A))$ 
6     else
7        $(P, A) \leftarrow \text{InternalRule}(J, (P, A))$ 
8   until  $((P, A) = (P', A')) \parallel \text{timeout}$ 
9 return  $(P, A)$ 

```

Algorithm 3. $\text{k-assume}[\mathcal{I}]((P, A))$

```

1 repeat
2    $(P', A') \leftarrow (P, A)$ 
3   foreach  $u \in \mathcal{U}$  such that  $P(u) = *$  do
4      $(P_0, A_0) \leftarrow (P, A)$ 
5      $(B, \overline{B}) \leftarrow (\top_{\mathcal{P}}[u \mapsto 0], \top_{\mathcal{P}}[u \mapsto 1])$ 
6      $(P_1, A_1) \leftarrow (P_0, A_0) \sqcap (B, \top)$ 
7      $(P_2, A_2) \leftarrow (P_0, A_0) \sqcap (\overline{B}, \top)$ 
8      $(P'_1, A'_1) \leftarrow (\text{k-1-assume}[\mathcal{I}]((P_1, A_1)))$ 
9      $(P'_2, A'_2) \leftarrow (\text{k-1-assume}[\mathcal{I}]((P_2, A_2)))$ 
10     $(P, A) \leftarrow (P'_1, A'_1) \sqcup (P'_2, A'_2)$ 
11  until  $((P, A) = (P', A')) \parallel \text{timeout}$ 
12 return  $(P, A)$ 

```

validity (tautology) checking “a formula is either [provable with $k = 1$] or not a tautology at all!” [18, p. 227].

3 Algorithm for $\widetilde{\text{Assume}}[\varphi](A)$

This section presents our algorithm for computing $\widetilde{\text{Assume}}[\varphi](A) \in \mathcal{A}$, for $\varphi \in \mathcal{L}$. The assumptions of our framework are as follows:

1. There is a Galois connection $\mathcal{C} \xrightleftharpoons[\alpha]{\gamma} \mathcal{A}$ between \mathcal{A} and concrete domain \mathcal{C} .
2. There is an algorithm to perform the join of arbitrary elements of \mathcal{A} .
3. Given a literal $l \in \mathcal{L}$, there is an algorithm $\mu\tilde{\alpha}$ to compute a safe (overapproximating) “micro- $\tilde{\alpha}$ ”—i.e., $A' = \mu\tilde{\alpha}(l)$ such that $\gamma(A') \supseteq \llbracket l \rrbracket$.
4. There is an algorithm to perform the meet of an arbitrary element of \mathcal{A} with an arbitrary element of $\{\mu\tilde{\alpha}(l) \mid l \in \text{literal}(\mathcal{L})\}$.

Note that \mathcal{A} is allowed to have infinite descending chains; because $\widetilde{\text{Assume}}$ works from above, it is allowed to stop at any time, and the value in hand is an overapproximation of the most precise answer.

Alg. 1 presents the algorithm that computes $\widetilde{\text{Assume}}[\varphi](A)$ for $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$. Line 2 calls the function `integrity`, which converts φ into integrity constraints \mathcal{I} by assigning a fresh Boolean variable to each subformula of φ , using the rules described in Fig. 2. The variable u_φ corresponds to formula φ . We use \mathcal{U} to denote the set of Boolean variables created when converting φ to \mathcal{I} . Alg. 1 also uses a second abstract domain \mathcal{P} , each of whose elements represents a set of Boolean assignments in $\mathbb{P}(\mathcal{U} \rightarrow \{0, 1\})$. For simplicity, in this paper \mathcal{P} is the Cartesian domain $(\mathcal{U} \rightarrow \{0, 1, *\})_\perp$, but other more-expressive Boolean domains could be used [35].

On line 2 of Alg. 1, an element of \mathcal{P} is created in which u_φ is assigned the value 1, which asserts that φ is true. Alg. 1 is parameterized by the value of k (where $k \geq 0$). Let $\gamma_{\mathcal{I}}((P, A))$ denote $\gamma((P, A)) \cap \llbracket \mathcal{I} \rrbracket$. The call to `k-assume` on

$$\frac{\varphi := \ell \quad \ell \in \text{literal}(\mathcal{L})}{u_\varphi \Leftrightarrow \ell \in \mathcal{I}} \text{ LEAF} \quad \frac{\varphi := \varphi_1 \text{op} \varphi_2}{u_\varphi \Leftrightarrow (u_{\varphi_1} \text{op} u_{\varphi_2}) \in \mathcal{I}} \text{ INTERNAL}$$

Fig. 2. Rules used to convert a formula $\varphi \in \mathcal{L}$ into a set of integrity constraints \mathcal{I} . op represents any binary connective in \mathcal{L} , and $\text{literal}(\mathcal{L})$ is the set of atomic formulas and their negations.

line (3) returns (\tilde{P}, \tilde{A}) , which is a semantic reduction of (P, A) with respect to \mathcal{I} ; that is, $\gamma_{\mathcal{I}}((\tilde{P}, \tilde{A})) = \gamma_{\mathcal{I}}((P, A))$ and $(\tilde{P}, \tilde{A}) \sqsubseteq (P, A)$. In general, the greater the value of k , the more precise is the result computed by Alg. II. The next theorem states that Alg. II computes an over-approximation of $\text{Assume}[\varphi](A)$.

Theorem 1 ([36]). *For all $\varphi \in \mathcal{L}, A \in \mathcal{A}$, if $\tilde{A} = \widetilde{\text{Assume}}[\varphi](A)$, then $\gamma(\tilde{A}) \supseteq [\varphi] \cap \gamma(A)$, and $\tilde{A} \sqsubseteq A$.* \square

Alg. B presents the algorithm to compute k-assume, for $k \geq 1$. Given the integrity constraints \mathcal{I} , and the current abstract value (P, A) , k-assume[\mathcal{I}]((P, A)) returns an abstract value that is a semantic reduction of (P, A) with respect to \mathcal{I} . The crux of the computation is the inner loop body, lines (4)–(10), which implements an analog of the Dilemma Rule from Stålmarck’s method [33].

The steps of the Dilemma Rule are shown schematically in Fig. II(b). At line (3) of Alg. B, a Boolean variable u whose value is unknown is chosen. $B = \top_{\mathcal{P}}[u \mapsto 0]$ and its complement $\overline{B} = \top_{\mathcal{P}}[u \mapsto 1]$ are used to split the current abstract value (P_0, A_0) into two abstract values $(P_1, A_1) = (P, A) \sqcap (B, \top)$ and $(P_2, A_2) = (P, A) \sqcap (\overline{B}, \top)$, as shown in lines (6) and (7).

The calls to (k-1)-assume at lines (8) and (9) compute semantic reductions of (P_1, A_1) and (P_2, A_2) with respect to \mathcal{I} , which creates (P'_1, A'_1) and (P'_2, A'_2) , respectively. Finally, at line (10) (P'_1, A'_1) and (P'_2, A'_2) are merged by performing a join. (The result is labeled (P_3, A_3) in Fig. II(b).)

The steps of the Dilemma Rule (Fig. II(b)) are repeated until a fixpoint is reached, or some resource bound is exceeded. The next theorem states that k-assume[\mathcal{I}]((P, A)) computes a semantic reduction of (P, A) with respect to \mathcal{I} .

Theorem 2 ([36]). *For all $P \in \mathcal{P}$ and $A \in \mathcal{A}$, if $(P', A') = \text{k-assume}[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$.* \square

Alg. 2 describes the algorithm to compute 0-assume: given the integrity constraints \mathcal{I} , and an abstract value (P, A) , 0-assume[\mathcal{I}]((P, A)) returns an abstract value (P', A') that is a semantic reduction of (P, A) with respect to \mathcal{I} . It is in this algorithm that information is passed between the component abstract values $P \in \mathcal{P}$ and $A \in \mathcal{A}$ via *propagation rules*, like the ones shown in Figs. B and 4. In lines (4)–(7) of Alg. 2, these rules are applied by using a single integrity constraint in \mathcal{I} and the current abstract value (P, A) .

Given $J \in \mathcal{I}$ and (P, A) , the net effect of applying any of the propagation rules is to compute a semantic reduction of (P, A) with respect to $J \in \mathcal{I}$. The propagation rules used in Alg. 2 can be classified into two categories:

$$\frac{J = (u_1 \Leftrightarrow (u_2 \vee u_3)) \in \mathcal{I} \quad P(u_1) = 0}{(P \sqcap \top[u_2 \mapsto 0, u_3 \mapsto 0], A)} \text{ OR1}$$

$$\frac{J = (u_1 \Leftrightarrow (u_2 \wedge u_3)) \in \mathcal{I} \quad P(u_1) = 1}{(P \sqcap \top[u_2 \mapsto 1, u_3 \mapsto 1], A)} \text{ AND1}$$

Fig. 3. Boolean rules used by Alg. 2 in the call InternalRule($J, (P, A)$)

$$\frac{J = (u \Leftrightarrow l) \in \mathcal{I} \quad P(u) = 1}{(P, A \sqcap \mu\tilde{\alpha}_A(l))} \text{ PTOA-1} \quad \frac{J = (u \Leftrightarrow l) \in \mathcal{I} \quad P(u) = 0}{(P, A \sqcap \mu\tilde{\alpha}_A(\neg l))} \text{ PTOA-0}$$

$$\frac{J = (u \Leftrightarrow \ell) \in \mathcal{I} \quad A \sqcap \mu\tilde{\alpha}_A(l) = \perp_A}{(P \sqcap \top[u \mapsto 0], A)} \text{ ATOP-0}$$

Fig. 4. Rules used by Alg. 2 in the call LeafRule($J, (P, A)$)

1. Rules that apply on line (7) when J is of the form $p \Leftrightarrow (q \text{ op } r)$, shown in Fig. 3. Such an integrity constraint is generated from each internal subformula of formula φ . These rules compute a non-trivial semantic reduction of P with respect to J by only using information from P . For instance, rule AND1 says that if J is of the form $p \Leftrightarrow (q \wedge r)$, and p is 1 in P , then we can infer that both q and r must be 1. Thus, $P \sqcap \top[q \mapsto 1, r \mapsto 1]$ is a semantic reduction of P with respect to J . (See Ex. 1, step 1)
2. Rules that apply on line (5) when J is of the form $u \Leftrightarrow \ell$, shown in Fig. 4. Such an integrity constraint is generated from each leaf of the original formula φ . This category of rules can be further subdivided into
 - (a) Rules that propagate information from abstract value P to abstract value A ; viz., rules PTOA-0 and PTOA-1. For instance, rule PTOA-1 states that given $J = u \Leftrightarrow l$, and $P(u) = 1$, then $A \sqcap \mu\tilde{\alpha}(l)$ is a semantic reduction of A with respect to J . (See Ex. 1, steps 2 and 3)
 - (b) Rule ATOP-0, which propagates information from abstract value A to abstract value P . Rule ATOP-0 states that if $J = (u \Leftrightarrow \ell)$ and $A \sqcap \mu\tilde{\alpha}(l) = \perp_A$, then we can infer that u is false. Thus, the value of $P \sqcap \top[u \mapsto 0]$ is a semantic reduction of P with respect to J . (See Ex. 1, step 4)

Alg. 2 repeatedly applies the propagation rules until a fixpoint is reached, or some resource bound is reached. The next theorem states that 0-assume computes a semantic reduction of (P, A) with respect to \mathcal{I} .

Theorem 3 ([36]). *For all $P \in \mathcal{P}, A \in \mathcal{A}$, if $(P', A') = 0\text{-assume}[\mathcal{I}]((P, A))$, then $\gamma_{\mathcal{I}}((P', A')) = \gamma_{\mathcal{I}}((P, A))$ and $(P', A') \sqsubseteq (P, A)$.* \square

4 Instantiations

In this section, we describe instantiations of our framework for two logical-language/abstract-domain pairs: QF_BV/KS and QF_LRA/Polyhedra. We say

that an $\widehat{\text{Assume}}$ algorithm is *complete* for a logic \mathcal{L} and abstract domain \mathcal{A} if it is guaranteed to compute the best value $\widehat{\text{Assume}}[\varphi](A)$ for $\varphi \in \mathcal{L}$ and $A \in \mathcal{A}$. We give conditions under which the two instantiations are complete.

Bitvector Affine-Relation Domain (QF_BV/KS). King and Søndergaard [21] gave an algorithm for $\widehat{\alpha}$ for an abstract domain of Boolean affine relations. Elder et al. [12] extended the algorithm to arithmetic modulo 2^w (i.e., bitvectors of width w). Both algorithms work from below, making repeated calls on a SAT solver (King and Søndergaard) or an SMT solver (Elder et al.), performing joins to create increasingly better approximations of the desired answer. We call this family of domains KS, and call the (generalized) algorithm $\widehat{\alpha}_{\text{KS}}^\uparrow$.

Given a literal $l \in \text{QF_BV}$, we compute $\mu\widehat{\alpha}_{\text{KS}}(l)$ by invoking $\widehat{\alpha}_{\text{KS}}^\uparrow(l)$. That is, we harness $\widehat{\alpha}_{\text{KS}}^\uparrow$ in service of $\widehat{\text{Assume}}_{\text{KS}}$, but only for $\mu\widehat{\alpha}_{\text{KS}}$, which means that $\widehat{\alpha}_{\text{KS}}^\uparrow$ is only applied to literals. If an invocation of $\widehat{\alpha}_{\text{KS}}^\uparrow$ does not return an answer within a specified time limit, we use \top_{KS} .

Alg. 11 is not complete for QF_BV/KS. Let x be a bitvector of width 2, and let $\varphi = (x \neq 0 \wedge x \neq 1 \wedge x \neq 2)$. Thus, $\widehat{\text{Assume}}[\varphi](\top_{\text{KS}}) = \{x - 3 = 0\}$. The KS domain is not expressive enough to represent disequalities. For instance, $\mu\widehat{\alpha}(x \neq 0)$ equals \top_{KS} . Because Alg. 11 considers only a single integrity constraint at a time, we get $\widehat{\text{Assume}}[\varphi](\top_{\text{KS}}) = \mu\widehat{\alpha}(x \neq 0) \sqcap \mu\widehat{\alpha}(x \neq 1) \sqcap \mu\widehat{\alpha}(x \neq 2) = \top_{\text{KS}}$.

The current approach can be made complete for QF_BV/KS by making 0-assume consider multiple integrity constraints during propagation (in the limit, having to call $\mu\widehat{\alpha}(\varphi)$). For the affine subset of QF_BV, an alternative approach would be to perform a 2^w -way split on the KS value each time a disequality is encountered, where w is the bit-width—in effect, rewriting $x \neq 0$ to $(x = 1 \vee x = 2 \vee x = 3)$. Furthermore, if there is a $\mu\widehat{\text{Assume}}$ operation, then the second approach can be extended to handle all of QF_BV: $\mu\widehat{\text{Assume}}[\ell](A)$ would be used to take the current KS abstract value A and a literal ℓ , and return an over-approximation of $\widehat{\text{Assume}}[\ell](A)$. All these approaches would be prohibitively expensive. Our current approach, though theoretically not complete, works very well in practice (see §5).

Polyhedral Domain (QF_LRA/Polyhedra). The second instantiation that we implemented is for the logic QF_LRA and the polyhedral domain [8]. Because a QF_LRA disequality $t \neq 0$ can be normalized to $(t < 0 \vee t > 0)$, every literal l in a normalized QF_LRA formula is merely a half-space in the polyhedral domain. Consequently, $\mu\widehat{\alpha}_{\text{Polyhedra}}(l)$ is exact, and easy to compute. Furthermore, because of this precision, the $\widehat{\text{Assume}}$ algorithm is complete for QF_LRA/Polyhedra. In particular, if $k = |\varphi|$, then k-assume is sufficient to guarantee that $\widehat{\text{Assume}}[\varphi](A)$ returns $\widehat{\text{Assume}}[\varphi](A)$. For polyhedra, our implementation uses PPL [28].

The observation in the last paragraph applies in general: if $\mu\widehat{\alpha}_{\mathcal{A}}(l)$ is exact for all literals $l \in \mathcal{L}$, then Alg. 11 is complete for logic \mathcal{L} and abstract domain \mathcal{A} .

5 Experiments

Bitvector Affine-Relation Analysis (ARA). We compare two methods for computing the abstract transformers for the KS domain for ARA [21]:

Prog. name	Measures of size				$\hat{\alpha}^\uparrow$ Performance			
	instrs	CFGs	BBs	brs	WPDS	t/o	post*	query
finger	532	18	298	48	110.9	4	0.266	0.015
subst	1093	16	609	74	204.4	4	0.344	0.016
label	1167	16	573	103	148.9	2	0.344	0.032
chkdisk	1468	18	787	119	384.4	16	0.219	0.031
convert	1927	38	1013	161	289.9	9	1.047	0.062
route	1982	40	931	243	562.9	14	1.281	0.046
logoff	2470	46	1145	306	621.1	16	1.938	0.063
setup	4751	67	1862	589	1524.7	64	0.968	0.047

Fig. 5. WPDS experiments ($\hat{\alpha}^\uparrow$). The columns show the number of instructions (instrs); the number of procedures (CFGs); the number of basic blocks (BBs); the number of branch instructions (brs); the times, in seconds, for WPDS construction with $\hat{\alpha}_\text{KS}^\uparrow$ weights, running post*, and finding one-vocabulary affine relations at blocks that end with branch instructions (query). The number of basic blocks for which $\hat{\alpha}_\text{KS}^\uparrow$ -weight generation timed out is listed under “t/o”.

- the $\hat{\alpha}^\uparrow$ -based procedure described in Elder et al. [12].
- the $\tilde{\alpha}$ -based procedure described in this paper (“ $\tilde{\alpha}^\downarrow$ ”), instantiated for KS.

Our experiments were designed to answer the following questions:

1. How does the speed of $\tilde{\alpha}^\downarrow$ compare with that of $\hat{\alpha}^\uparrow$?
2. How does the precision of $\tilde{\alpha}^\downarrow$ compare with that of $\hat{\alpha}^\uparrow$?

To address these questions, we performed ARA on x86 machine code, computing affine relations over the x86 registers. Our experiments were run on a single core of a quad-core 3.0 GHz Xeon computer running 64-bit Windows XP (SP2), configured so that a user process has 4GB of memory. We analyzed a corpus of Windows utilities using the WALI [20] system for weighted pushdown systems (WPDSs). For the baseline $\hat{\alpha}^\uparrow$ -based analysis we used a weight domain of $\hat{\alpha}^\uparrow$ -generated KS transformers. The weight on each WPDS rule encodes the KS transformer for a basic block B of the program, including a jump or branch to a successor block. A formula φ_B is created that captures the concrete semantics of B , and then the KS weight for B is obtained by performing $\hat{\alpha}^\uparrow(\varphi_B)$ (cf. Ex. 11). We used EWPDS merge functions [21] to preserve caller-save and callee-save registers across call sites. The post* query used the FWPDs algorithm [23].

Fig. 5 lists several size parameters of the examples (number of instructions, procedures, basic blocks, and branches) along with the times for constructing abstract transformers and running post*². Col. 6 of Fig. 5 shows that the calls to $\hat{\alpha}^\uparrow$ during WPDS construction dominate the total time for ARA.

Each call to $\hat{\alpha}^\uparrow$ involves repeated invocations of an SMT solver. Although the overall time taken by $\hat{\alpha}^\uparrow$ is not limited by a timeout, we use a 3-second timeout for each invocation of the SMT solver (as in Elder et al. [12]). Fig. 5 lists the number of such SMT solver timeouts for each benchmark. In case the invocation

² Due to the high cost of the $\hat{\alpha}^\uparrow$ -based WPDS construction, all analyses excluded the code for libraries. Because register `eax` holds the return value from a call, library functions were modeled approximately (albeit unsoundly, in general) by “`havoc(eax)`”.

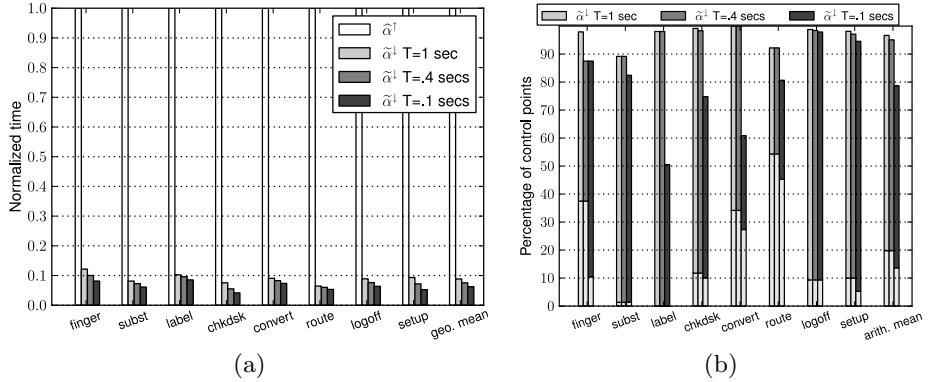


Fig. 6. (a) Performance: $\tilde{\alpha}^\downarrow$ vs. $\hat{\alpha}^\uparrow$. (b) Precision: % of control points at which $\tilde{\alpha}^\downarrow$ has as good or better precision as $\hat{\alpha}^\uparrow$; the lighter-color lower portion of each bar indicates the % of control points at which the precision is strictly greater for $\tilde{\alpha}^\downarrow$.

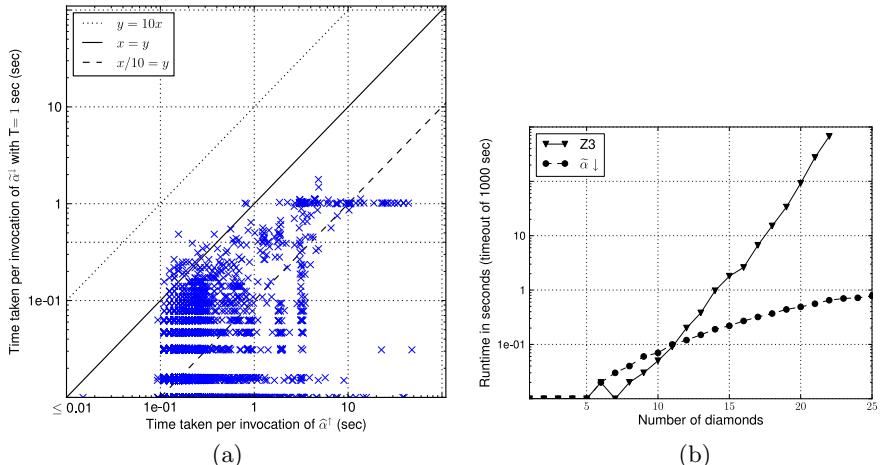


Fig. 7. (a) Log-log scatter plot of transformer-construction time. (b) Semilog plot of Z3 vs. $\tilde{\alpha}^\downarrow$ on χ_d formulas.

of the SMT solver times out, $\hat{\alpha}^\uparrow$ is forced to return \top_{KS} in order to be sound. (Consequently, it is possible for $\tilde{\alpha}^\downarrow$ to return a more precise answer than $\hat{\alpha}^\uparrow$.)

The setup for the $\tilde{\alpha}^\downarrow$ -based analysis is the same as the baseline $\hat{\alpha}^\uparrow$ -based analysis, except that we call $\tilde{\alpha}^\downarrow$ when calculating the KS weight for a basic block. We use 1-assume in this experiment. Each basic-block formula φ_B is rewritten to a set of integrity constraints, with ITE-terms rewritten as illustrated in Ex. II. The priority of a Boolean variable is its postorder-traversal number, and is used to select which variable is used in the Dilemma Rule. We bound the total time taken by each call to $\tilde{\alpha}^\downarrow$ to a fixed timeout T . Note that even when the call to $\tilde{\alpha}^\downarrow$ times out, it can still return a sound non- \top_{KS} value. We ran $\tilde{\alpha}^\downarrow$ using $T = 1$ sec, $T = 0.4$ secs, and $T = 0.1$ secs.

Fig. 6(a) shows the normalized time taken for WPDS construction when using $\tilde{\alpha}^\downarrow$ with $T = 1$ sec, $T = 0.4$ secs, and $T = 0.1$ secs. The running time is normalized to the corresponding time taken by $\hat{\alpha}^\uparrow$; lower numbers are better. WPDS construction using $\tilde{\alpha}^\downarrow$ with $T = 1$ sec. is about 11.3 times faster than $\hat{\alpha}^\uparrow$ (computed as the geometric mean), which answers question 1.

Decreasing the timeout T makes the $\tilde{\alpha}^\downarrow$ WPDS construction only slightly faster: on average, going from $T = 1$ sec. to $T = .4$ secs. reduces WPDS construction time by only 17% (computed as the geometric mean). To understand this behavior better, we show in Fig. 7(a) a log-log scatter-plot of the times taken by $\hat{\alpha}^\uparrow$ versus the times taken by $\tilde{\alpha}^\downarrow$ (with $T = 1$ sec.), to generate the transformers for each basic block in the benchmark suite. As shown in Fig. 7(a), the times taken by $\tilde{\alpha}^\downarrow$ are bounded by 1 second. (There are a few calls that take more than 1 second; they are an artifact of the granularity of operations at which we check whether the procedure has timed out.) Most of the basic blocks take less than 0.4 seconds, which explains why the overall time for WPDS construction does not decrease much as we decrease T in Fig. 6(a). We also see that the $\hat{\alpha}^\uparrow$ times are not bounded, and can be as high as 50 seconds.

To answer question 2 we compared the precision of the WPDS analysis when using $\tilde{\alpha}^\downarrow$ with T equal to 1, 0.4, and 0.1 seconds with the precision obtained using $\hat{\alpha}^\uparrow$. In particular, we compare the affine relations (i.e., invariants) computed by the $\tilde{\alpha}^\downarrow$ -based and $\hat{\alpha}^\uparrow$ -based analyses for each *control point*—i.e., the beginning of a basic block that ends with a branch. Fig. 6(b) shows the percentage of control points for which the $\tilde{\alpha}^\downarrow$ -based analysis computes a better (tighter) or equally precise affine relation. On average, when using $T = 1$ sec, $\tilde{\alpha}^\downarrow$ -based analysis computes an *equally precise* invariant at 76.9% of the control points (computed as the arithmetic mean). Interestingly, the $\tilde{\alpha}^\downarrow$ -based analysis computes an answer that is *more precise* compared to that computed by the $\hat{\alpha}^\uparrow$ -based analysis. That is not a bug in our implementation; it happens because $\hat{\alpha}^\uparrow$ has to return \top_{KS} when the call to the SMT solver times out. In Fig. 6(b), the lighter-color lower portion of each bar shows the percentage of control points for which $\tilde{\alpha}^\downarrow$ -based analysis provides strictly more precise invariants when compared to $\hat{\alpha}^\uparrow$ -based analysis; on average, $\tilde{\alpha}^\downarrow$ -based analysis is more precise for 19.8% of the control points (arithmetic mean, for $T = 1$ second). $\tilde{\alpha}^\downarrow$ -based analysis is less precise at only 3.3% of the control points. Furthermore, as expected, when the timeout for $\tilde{\alpha}^\downarrow$ is reduced, the precision decreases.

Satisfiability Checking. The formula used in Ex. 2 is just one instance of a family of unsatisfiable QF_LRA formulas [25]. Let $\chi_d = (a_d < a_0) \wedge \bigwedge_{i=0}^{d-1} ((a_i < b_i) \wedge (a_i < c_i) \wedge ((b_i < a_{i+1}) \vee (c_i < a_{i+1})))$. The formula ψ in Ex. 2 is χ_2 ; that is, the number of “diamonds” is 2 (see Fig. 1(a)). We used the QF_LRA/Polyhedra instantiation of our framework to check whether $\tilde{\alpha}(\chi_d) = \perp$ for $d = 1 \dots 25$ using 1-assume. We ran this experiment on a single processor of a 16-core 2.4 GHz Intel Zeon computer running 64-bit RHEL Server release 5.7. The semilog plot in Fig. 7(b) compares the running time of $\tilde{\alpha}^\downarrow$ with that of Z3, version 3.2 [11]. The time taken by Z3 increases exponentially with d , exceeding the timeout threshold of 1000 seconds for $d = 23$. This corroborates the results of a

similar experiment conducted by McMillan et al. [25], where the reader can also find an in-depth explanation of this behavior.

On the other hand, the running time of $\tilde{\alpha}^\downarrow$ increases linearly with d taking 0.78 seconds for $d = 25$. The cross-over point is $d = 12$. In Ex. 2 we saw how two successive applications of the Dilemma Rule suffice to prove that ψ is unsatisfiable. That explanation generalizes to χ_d : d applications of the Dilemma Rule are sufficient to prove unsatisfiability of χ_d . The order in which Boolean variables with unknown truth values are selected for use in the Dilemma Rule has no bearing on this linear behavior, as long as no variable is starved from being chosen (i.e., a fair-choice schedule is used). Each application of the Dilemma Rule is able to infer that $a_i < a_{i+1}$ for some i .

We do not claim that $\tilde{\alpha}^\downarrow$ is better than mature SMT solvers such as Z3. We do believe that it represents another interesting point in the design space of SMT solvers, similar in nature to the GDPPLL algorithm [25] and the k -lookahead technique used in the DPLL(\sqcup) algorithm [4].

6 Applications to Other Symbolic Operations

The symbolic operations of $\widehat{\gamma}$ and $\widehat{\alpha}$ can be used to implement a number of other useful operations, as discussed below. In each case, over-approximations result if $\widehat{\alpha}$ is replaced by $\widetilde{\alpha}$.

- The operation of containment checking, $A_1 \sqsubseteq A_2$, which is needed by analysis algorithms to determine when a post-fixpoint is attained, can be implemented by checking whether $\widehat{\alpha}(\widehat{\gamma}(A_1) \wedge \neg\widehat{\gamma}(A_2))$ equals \perp .
- Suppose that there are two Galois connections $\mathcal{G}_1 = \mathcal{C} \xrightleftharpoons[\alpha_1]{\gamma_1} \mathcal{A}_1$ and $\mathcal{G}_2 = \mathcal{C} \xrightleftharpoons[\alpha_2]{\gamma_2} \mathcal{A}_2$, and one wants to work with the reduced product of \mathcal{A}_1 and \mathcal{A}_2 [7, §10.1]. The semantic reduction of a pair (A_1, A_2) can be performed by letting ψ be the formula $\widehat{\gamma}_1(A_1) \wedge \widehat{\gamma}_2(A_2)$, and creating the pair $(\widehat{\alpha}_1(\psi), \widehat{\alpha}_2(\psi))$.
- Given $A_1 \in \mathcal{A}_1$, one can find the most-precise value $A_2 \in \mathcal{A}_2$ that over-approximates A_1 in \mathcal{A}_2 as follows: $A_2 = \widehat{\alpha}_2(\widehat{\gamma}_1(A_1))$.
- Given a loop-free code fragment F , consisting of one or more blocks of program statements and conditions, one can obtain a representation of its best transformer by symbolically executing F to obtain a transition formula ψ_F , and then performing $\widehat{\alpha}(\psi_F)$.

7 Related Work

Extensions of Stålmarck’s Method. Björk [3] describes extensions of Stålmarck’s method to first-order logic. Like Björk, our work goes beyond the classical setting of Stålmarck’s method [33] (i.e., propositional logic) and extends the method to more expressive logics, such as QF_LRA or QF_BV. However, Björk is concerned solely with validity checking, and—compared with the

propositional case—the role of abstraction is less clear in his method. Our algorithm not only uses an abstract domain as an explicit datatype, the goal of the algorithm is to compute an abstract value $A' = \widetilde{\text{Assume}}[\varphi](A)$.

Our approach was influenced by Granger’s method of using (in)equation solving as a way to implement semantic reduction and Assume as part of his technique of *local decreasing iterations* [16]. Granger describes techniques for performing reductions with respect to (in)equations of the form $x_1 \bowtie F(x_1, \dots, x_n)$ and $(x_1 * F(x_1, \dots, x_n)) \bowtie G(x_1, \dots, x_n)$, where \bowtie stands for a single relational symbol of \mathcal{L} , such as $=$, \neq , $<$, \leq , $>$, \geq , or \equiv (arithmetical congruence). Our framework is not limited to literals of these forms; all that we require is that for a literal $l \in \mathcal{L}$, there is an algorithm to compute an overapproximating value $\mu\tilde{\alpha}(l)$. Moreover, Granger has no analog of the Dilemma Rule, nor does he present any completeness results (cf. §1).

SMT Solvers. Most methods for SMT solving can be classified according to whether they employ *lazy* or *eager* translations to SAT. (The SAT procedure then employed is generally based on the DPLL procedure [10, 9].) In contrast, the algorithm for SMT described in this paper is not based on a translation to SAT; instead, it generalizes Stålmarck’s method for propositional logic to richer logics.

Lazy approaches abstract each atom of the input formula to a distinct propositional variable, use a SAT solver to find a propositional model, and then check that model against the theory [11, 13, 11]. The disadvantage of the lazy approach is that it cannot use theory information to prune the search. In contrast, our algorithm is able to use theory-specific information to make deductions—in particular, in the LeafRule function (Fig. 4) used in Alg. 2. The use of theory-specific information is the reason why our approach outperformed Z3, which uses the lazy approach, on the diamond example (§5).

Eager approaches [5, 34] encode more of the theory into the propositional formula that is given to the SAT solver, and hence are able to constrain the solution space with theory-specific information. The challenge in designing such solvers is to ensure that the propositional formula does not blow up in size. In our approach, such an explosion in the set of literals in the formula is avoided because our learned facts are restricted by the abstract domain in use.

A variant of the Dilemma Rule is used in DPLL(\sqcup), and allows the theory solver in a lazy DPLL-based SMT solver to produce joins of facts deduced along different search paths. However, as pointed out by Bjørner et al. [4, §5], their system is weaker than Stålmarck’s method, because Stålmarck’s method can learn equivalences between literals.

Another difference between our work and existing approaches to SMT is the connection presented in this paper between Stålmarck’s method and the computation of best abstract operations for abstract interpretation.

Best Abstract Operations. Several papers about best abstract operations have appeared in the literature [4, 29, 37, 21, 12]. Graf and Saïdi [4] showed that decision procedures can be used to generate best abstract transformers for predicate-abstraction domains. Other work has investigated more efficient

methods to generate approximate transformers that are not best transformers, but approach the precision of best transformers [26].

Several techniques work from *below* [29, 21, 12]—performing joins to incorporate more and more of the concrete state space—which has the drawback that if they are stopped before the final answer is reached, the most-recent approximation is an *under-approximation* of the desired value. In contrast, our technique works from *above*. It can stop at any time and return a safe answer.

Yorsh et al. [37] developed a method that works from above to perform $\widetilde{\text{Assume}}[\varphi](A)$ for the kind of abstract domains used in shape analysis (i.e., “canonical abstraction” of logical structures [30]). Their method has a splitting step, but no analog of the join step performed at the end of an invocation of the Dilemma Rule. In addition, their propagation rules are much more heavyweight.

Template Constraint Matrices (TCMs) are a parametrized family of linear-inequality domains for expressing invariants in linear real arithmetic. Sankaranarayanan et al. [31] gave a parametrized meet, join, and set of abstract transformers for all TCM domains. Monniaux [26] gave an algorithm that finds the best transformer in a TCM domain across a straight-line block (assuming that concrete operations consist of piecewise linear functions), and good transformers across more complicated control flow. However, the algorithm uses quantifier elimination, and no polynomial-time elimination algorithm is known for piecewise-linear systems.

Cover Algorithms. Gulwani and Musuvathi [17] defined the “cover problem”, which addresses *approximate existential quantifier elimination*: Given a formula φ in logic \mathcal{L} , and a set of variables V , find the strongest quantifier-free formula $\overline{\varphi}$ in \mathcal{L} such that $[\exists V : \varphi] \subseteq [\overline{\varphi}]$. They presented cover algorithms for the theories of uninterpreted functions and linear arithmetic, and showed that covers exist in some theories that do not support quantifier elimination.

The notion of a cover has similarities to the notion of symbolic abstraction, but the two notions are distinct. Our technical report [36] discusses the differences in detail, describing symbolic abstraction as over-approximating a formula φ using an impoverished logic fragment, while a cover algorithm only removes variables V from the vocabulary of φ . The two approaches yield different over-approximations of φ , and the over-approximation obtained by a cover algorithm does not, in general, yield suitable abstract values and abstract transformers.

References

1. Armando, A., Castellini, C., Giunchiglia, E.: SAT-Based Procedures for Temporal Reasoning. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 97–108. Springer, Heidelberg (2000)
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
3. Björk, M.: First order Stålmarck. J. Autom. Reasoning 42(1), 99–122 (2009)
4. Bjørner, N., de Moura, L.: Accelerated lemma learning using joins–DPLL(\sqcup). In: LPAR (2008)

5. Bryant, R.E., Velev, M.N.: Boolean satisfiability with transitivity constraints. *Trans. on Computational Logic* 3(4) (2002)
6. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. *FMSD* 25(2-3) (2004)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: *POPL* (1979)
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear constraints among variables of a program. In: *POPL* (1978)
9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. *Commun. ACM* 5(7) (1962)
10. Davis, M., Putnam, H.: A computing procedure for quantification theory. *J. ACM* 7(3) (1960)
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)
12. Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T.: Abstract Domains of Affine Relations. In: Yahav, E. (ed.) *SAS 2011. LNCS*, vol. 6887, pp. 198–215. Springer, Heidelberg (2011)
13. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem Proving Using Lazy Proof Explication. In: Hunt Jr., W.A., Somenzi, F. (eds.) *CAV 2003. LNCS*, vol. 2725, pp. 355–367. Springer, Heidelberg (2003)
14. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) *CAV 1997. LNCS*, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
15. Graham, S., Wegman, M.: A fast and usually linear algorithm for data flow analysis. *J. ACM* 23(1), 172–202 (1976)
16. Granger, P.: Improving the Results of Static Analyses Programs by Local Decreasing Iteration. In: Shyamasundar, R.K. (ed.) *FSTTCS 1992. LNCS*, vol. 652, pp. 68–79. Springer, Heidelberg (1992)
17. Gulwani, S., Musuvathi, M.: Cover Algorithms and Their Combination. In: Gairing, M. (ed.) *ESOP 2008. LNCS*, vol. 4960, pp. 193–207. Springer, Heidelberg (2008)
18. Harrison, J.: Stålmarck’s Algorithm as a HOL Derived Rule. In: von Wright, J., Harrison, J., Grundy, J. (eds.) *TPHOLs 1996. LNCS*, vol. 1125, pp. 221–234. Springer, Heidelberg (1996)
19. Karr, M.: Affine relationship among variables of a program. *Acta Inf.* 6 (1976)
20. Kidd, N., Lal, A., Reps, T.: WALi: The Weighted Automaton Library (2007), www.cs.wisc.edu/wpis/wpds/download.php
21. King, A., Søndergaard, H.: Automatic Abstraction for Congruences. In: Barthe, G., Hermenegildo, M. (eds.) *VMCAI 2010. LNCS*, vol. 5944, pp. 197–213. Springer, Heidelberg (2010)
22. Knoop, J., Steffen, B.: Interprocedural Coincidence Theorem. In: Pfahler, P., Kasstens, U. (eds.) *CC 1992. LNCS*, vol. 641, pp. 125–140. Springer, Heidelberg (1992)
23. Lal, A., Reps, T.: Improving Pushdown System Model Checking. In: Ball, T., Jones, R.B. (eds.) *CAV 2006. LNCS*, vol. 4144, pp. 343–357. Springer, Heidelberg (2006)
24. Lal, A., Reps, T., Balakrishnan, G.: Extended Weighted Pushdown Systems. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005. LNCS*, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)
25. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to Richer Logics. In: Bouajjani, A., Maler, O. (eds.) *CAV 2009. LNCS*, vol. 5643, pp. 462–476. Springer, Heidelberg (2009)
26. Monniaux, D.: Automatic modular abstractions for template numerical constraints. *LMCS* 6(3) (2010)

27. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. TOPLAS (2007)
28. PPL: The Parma polyhedra library, <http://www.cs.unipr.it/ppl/>
29. Reps, T., Sagiv, M., Yorsh, G.: Symbolic Implementation of the Best Transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
30. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS 24(3), 217–298 (2002)
31. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
32. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. Prentice-Hall (1981)
33. Sheeran, M., Stålmarck, G.: A tutorial on Stålmarck’s proof procedure for propositional logic. FMSD 16(1) (2000)
34. Strichman, O.: On Solving Presburger and Linear Arithmetic with SAT. In: Aagaard, M.D., O’Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 160–170. Springer, Heidelberg (2002)
35. Thakur, A., Reps, T.: A generalization of Stålmarck’s method. TR 1699. CS Dept., Univ. of Wisconsin, Madison, WI (October 2011)
36. Thakur, A., Reps, T.: A method for symbolic computation of precise abstract operations. TR 1708. CS Dept., Univ. of Wisconsin, Madison, WI (January 2012)
37. Yorsh, G., Reps, T., Sagiv, M.: Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 530–545. Springer, Heidelberg (2004)

Leveraging Interpolant Strength in Model Checking^{*}

Simone Fulvio Rollini¹, Ondrej Sery^{1,2}, and Natasha Sharygina¹

¹ Formal Verification Lab, University of Lugano, Switzerland

{simone.fulvio.rollini,ondrej.sery,natasha.sharygina}@usi.ch

<http://verify.inf.usi.ch>

² Dept. of Distributed and Dependable Systems, Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

<http://d3s.mff.cuni.cz>

Abstract. Craig interpolation is a well known method of abstraction successfully used in both hardware and software model checking. The logical strength of interpolants can affect the quality of approximations and consequently the performance of the model checkers. Recently, it was observed that for the same resolution proof a complete lattice of interpolants ordered by strength can be derived. Most state-of-the-art model checking techniques based on interpolation subject the interpolants to constraints that ensure efficient verification as, for example, in transition relation approximation for bounded model checking, counterexample-guided abstraction refinement and function summarization for software update checking. However, in general, these verification-specific constraints are not satisfied by all possible interpolants.

The paper analyzes the restrictions within the lattice of interpolants under which the required constraints are satisfied. This enables investigation of the effect of the strength of interpolants on the particular techniques, while preserving their soundness. As an additional benefit, combination of this result with proof manipulation procedures allows the use of optimized solvers to generate interpolants of different strengths for various model checking techniques.

1 Introduction

Craig interpolants [4] are commonly used for abstraction in hardware and software model checking. Recently, it was shown [5] that for the same resolution proof a complete lattice of interpolants ordered by the implication relation, i.e., *strength*, can be systematically derived. The strength of the interpolants may influence speed of convergence of the model checking algorithms as well as the amount of spurious behaviors that require refinement. The result in [5] shows that there are interpolants of different strengths to choose from. However, [5]

* This work is partially supported by the European Community under the call FP7-ICT-2009-5 — project PINCETTE 257647. The work of the second author was partially supported by the Grant Agency of the Czech Republic project P202/12/P180.

opens two new research problems. First, it is not clear how to choose the right interpolation algorithm for a particular model checking application. Second, if a concrete application puts additional constraints on the interpolants, it is unclear if the choice among interpolants of various strengths is restricted and how much.

This paper presents a theoretical solution to the second problem. We identify two classes of common interpolation-based model checking approaches that indeed put additional requirements on the interpolants. Then we formally determine and prove the restrictions for both classes on the choice of the interpolants strength under which these requirements are satisfied.

The first class of approaches concerns simultaneous abstraction by multiple interpolants. In this scenario, we have an unsatisfiable formula in the form of a conjunction of subformulae. From the proof of unsatisfiability, we compute interpolants abstracting the different conjuncts. The additional requirement (*Req1*) is ensuring unsatisfiability of the original formula with multiple conjuncts replaced by the corresponding interpolants. A notable example of this setting is the approach presented in [8], where the abstract transition relation is iteratively refined using interpolants. The authors notice the requirement imposed on the interpolants, and observe it satisfied while implicitly assuming the use of the interpolation algorithm of [11]. However, [8] is restricted to a single interpolant generated by this algorithm. Our solution overcomes this limitation by showing formally how to generate interpolants of different strength that satisfy the requirement. Interestingly, we discovered that not all interpolants do. Another application is software update checking, where the formula represents the original program with different conjuncts representing different functions as, e.g., in [17]. When a subset of functions is updated due to code changes and fixes, this approach checks if the interpolants remain valid abstractions of the new function bodies. This is a local check to show that a formula representing the new function body still implies the corresponding interpolant. If the check succeeds, unsatisfiability of the formula with multiple conjuncts replaced by the corresponding interpolants (*Req1*) provides correctness of the updated system without the need to check the entire formula of the updated system again.

The second class of model checking methods is typical of counterexample-guided abstraction refinement (CEGAR) [3]. Given a spurious error trace, the goal is to annotate nodes of an abstract reachability tree with an inductive sequence of formulae that together rule out the trace. The spurious error trace is represented by an unsatisfiable path formula, constructed from the SSA form of the trace. An interpolant is computed from the prefix and suffix of the trace for every location along the error trace. Here, the additional requirement (*Req2*) is that the resulting sequence of interpolants is inductive, i.e., that for every location, the current interpolant conjuncted with the precise representation of the instruction at the location implies the next interpolant along the trace. For example, this reasoning is crucial for the refinement techniques used in BLAST [1], IMPACT [13] and WOLVERINE [9]. In general, however, such a sequence is not inductive. Therefore, the authors restrict themselves to specific proof systems to ensure this property [7] ruling out not only the choice of interpolants of varying

strength but also the possibility of using state-of-the-art solvers. Other authors even require multiple solver calls to ensure a similar property [6].

Contribution. The main contribution is a theoretical formulation and proof of the constraints within the lattice of interpolants, such that Req1 and Req2 are satisfied.

In particular, building on [5], this work analyzes a whole family of interpolation procedures from which the lattice is generated. The analysis yields two interesting results: (1) we prove that every member of the family that produces interpolants stronger than the ones given by Pudlák’s algorithm [15] complies with Req1, and (2) we identify a subset, within the family of procedures (by means of the logical constraints that characterize it), that satisfies Req2. These results allow for the systematic study of how interpolants strength affects state-of-the-art model checking techniques, while preserving their soundness.

Since our results are not limited to the use of an ad-hoc proof system, any state-of-the-art solver can be chosen to generate proofs (if needed, post-processed by, e.g., the techniques of [2]) from which the interpolants are computed. Additionally, proof manipulation procedures, as in [16], can be applied to alter the size and the strength of interpolants in the various model checking applications.

2 Preliminaries

2.1 Craig Interpolation

Craig interpolants [4], since the seminal work by McMillan [10], have been extensively applied in SAT-based model checking and predicate abstraction [12]. Formally, given an unsatisfiable conjunction of formulae $A \wedge B$, an interpolant I is a formula that is implied by A (i.e., $A \rightarrow I$), is unsatisfiable in conjunction with B (i.e., $B \wedge I \rightarrow \perp$) and is defined on the common language of A and B . The interpolant I can be thought of as an over-approximation of A that still conflicts with B .

Several state-of-the art approaches exist to generate interpolants in an automated manner; the most successful techniques derive an interpolant for $A \wedge B$ (in certain proof systems) from a proof of unsatisfiability of the conjunction. This approach grants two important benefits: the generation can be achieved in linear time w.r.t. the proof size, and interpolants themselves only contain information relevant to determine the unsatisfiability of $A \wedge B$. In particular, Pudlák [15] investigates interpolation in the context of resolution systems for propositional logic, while McMillan [11] addresses both propositional logic and a quantifier free combination of the theories of uninterpreted functions and linear arithmetic. All these techniques adopt recursive algorithms, which initially set *partial interpolants* for the axioms. Then, following the proof structure, they deduce a partial interpolant for each conclusion from those of the premises. The partial interpolant of the overall conclusion is the interpolant for the formula.

2.2 Resolution Proofs

Assuming a finite set of propositional variables, a literal is a variable, either with positive (p) or negative (\bar{p}) polarity. A clause C is a finite disjunction of literals; a formula φ in conjunctive normal form (CNF) is a finite conjunction of clauses. A *resolution proof of unsatisfiability* (or *refutation*) of a formula ϕ in CNF is a tree such that the leaves are the clauses of ϕ , the root is the empty clause \perp and the inner nodes are clauses generated by means of the *resolution rule*:

$$\frac{C^+ \vee p \quad C^- \vee \bar{p}}{C^+ \vee C^-}$$

where $C^+ \vee p$ and $C^- \vee \bar{p}$ are the *antecedents*, $C^+ \vee C^-$ the *resolvent* and p is the *pivot* of the resolution step.

2.3 Strength of Interpolants

D'Silva et al. [5] generalize the algorithms by Pudlák [15] and McMillan [11] (as well as the approach dual to McMillan's, which we will call McMillan') for propositional resolution systems by introducing the notion of *labeled interpolation system*, focusing on the concept of *interpolant strength* (a formula ϕ is stronger than χ whenever $\phi \rightarrow \chi$). They present an analysis and a comparison of the systems corresponding to the three algorithms, together with a method to combine labeled systems in order to obtain weaker or stronger interpolants from a given proof of unsatisfiability. Throughout the paper we will adopt the notation of [5], adapted as necessary.

Given a refutation of a formula $A \wedge B$, a variable p can appear as literal only in A , only in B or in both conjuncts; p is respectively said to have *class* A , B or AB . The authors define a *labeling* L as a mapping that assigns a *color* among $\{a, b, ab\}$ independently to each variable in each clause (since a variable cannot have two occurrences in a clause, this is equivalent to coloring literals). The set of possible labelings is restricted by ensuring that class A variables receive color a and class B variables receive color b ; freedom is left for AB variables to be colored either a , b or ab .

In [5], a *labeled interpolation system* is defined as a procedure Itp (shown in Table 1) that, given A, B , a refutation R of $A \wedge B$ and a labeling L , outputs a partial interpolant $Itp_L(A, B, R, C)$ for any clause C in R ; this depends on the clause being in A or B (if leaf) and on the color of the pivot associated with the resolution step (if inner node). $Itp_L(A, B, R)$ represents the interpolant for $A \wedge B$, that is $Itp_L(A, B, R) \triangleq Itp_L(A, B, R, \perp)$ ¹. We will omit the parameters whenever clear from the context.

In Table 1, $C \downarrow \alpha$ denotes the restriction of a clause C to the literals of color α . $p : \alpha$ indicates that variable p has color α . By $C[I]$ we represent that clause C has a partial interpolant I . I^+ , I^- and I are the partial interpolants respectively associated with the two antecedents and the resolvent of a resolution step: $I^+ \triangleq Itp_L(C^+ \vee p)$, $I^- \triangleq Itp_L(C^- \vee \bar{p})$, $I \triangleq Itp_L(C^+ \vee C^-)$.

¹ As customary, we use \triangleq to characterize a definition, while \equiv a correspondence.

Table 1. Labeled interpolation system Itp_L

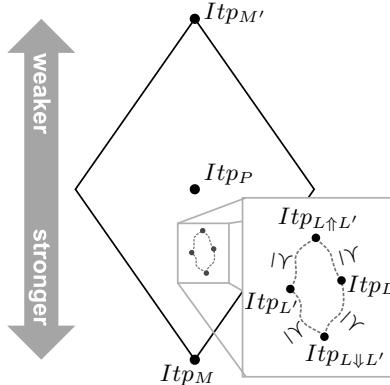
Leaf:	$C[I]$
$I = \begin{cases} C \downarrow b & \text{if } C \in A \\ \neg(C \downarrow a) & \text{if } C \in B \end{cases}$	

Inner node:	$C^+ \vee p : \alpha [I^+]$	$C^- \vee \bar{p} : \beta [I^-]$
	$C^+ \vee C^- [I]$	
$I = \begin{cases} I^+ \vee I^- & \text{if } \alpha \sqcup \beta = a \\ I^+ \wedge I^- & \text{if } \alpha \sqcup \beta = b \\ (I^+ \vee p) \wedge (I^- \vee \bar{p}) & \text{if } \alpha \sqcup \beta = ab \end{cases}$		

Table 2. Pudlák's interpolation system Itp_P

Leaf:	$C[I]$
$I = \begin{cases} \perp & \text{if } C \in A \\ \top & \text{if } C \in B \end{cases}$	

Inner node:	$C^+ \vee p : \alpha [I^+]$	$C^- \vee \bar{p} : \alpha [I^-]$
	$C^+ \vee C^- [I]$	
$I = \begin{cases} I^+ \vee I^- & \text{if } \alpha = a \\ I^+ \wedge I^- & \text{if } \alpha = b \\ (I^+ \vee p) \wedge (I^- \vee \bar{p}) & \text{if } \alpha = ab \end{cases}$		

**Fig. 1.** Lattice of labeled interpolation systems

An operator \sqcup allows to determine the color of a pivot p , taking into account that p might have different colors α and β in the two antecedents: \sqcup is idempotent, symmetric and defined by $a \sqcup b \triangleq ab$, $a \sqcup ab \triangleq ab$, $b \sqcup ab \triangleq ab$.

The systems corresponding to McMillan, Pudlák and McMillan's interpolation algorithms will be referred to as Itp_M , Itp_P , $Itp_{M'}$. Itp_L subsumes Itp_M , Itp_P and $Itp_{M'}$, obtained as special cases by coloring all the occurrences of AB variables with b , ab and a , respectively (compare, for example, Tables 1 and 2).

A total order \preceq is defined over the colors as $b \preceq ab \preceq a$, and extended to a partial order over labeled systems: $Itp_L \preceq Itp_{L'}$ if, for every clause C and variable p in C , $L(p, C) \preceq L'(p, C)$. This allows the authors to directly compare the logical strength of the interpolants produced by two systems. In fact, for any refutation R of a formula $A \wedge B$ and labelings L, L' such that $L \preceq L'$, we have: $Itp_L(A, B, R) \rightarrow Itp_{L'}(A, B, R)$ and we say that Itp_L is *stronger* than $Itp_{L'}$.

Two interpolation systems Itp_L and $Itp_{L'}$ can generate new systems $Itp_{L\uparrow L'}$ and $Itp_{L\downarrow L'}$ by combining the labelings L and L' in accordance with a relation \preceq : $(L \uparrow L')(p, C) \triangleq \max_{\preceq}\{L(p, C), L'(p, C)\}$ and $(L \downarrow L')(p, C) \triangleq \min_{\preceq}\{L(p, C), L'(p, C)\}$. The authors remark that the collection of labeled systems over a refutation, together with the order \preceq and the operators \uparrow, \downarrow , represent a *complete lattice*, where Itp_M is the greatest element and $Itp_{M'}$ is the least, with Itp_P being in between (see Fig. 1).

3 Simultaneous Abstraction with Interpolation

This section analyzes simultaneous abstraction of the conjuncts of an unsatisfiable formula by means of multiple interpolants. The requirement (Req1) is to guarantee that the formula obtained by replacing the conjuncts with the respective interpolants remains unsatisfiable². We formally describe the problem, proving that the requirement is satisfied if the interpolants are generated using Pudlák's interpolation system, and later generalize the result to any interpolation system which is stronger than Pudlák's. We conclude by illustrating the applications to model checking.

3.1 Problem Description

As input, we assume an unsatisfiable formula ϕ in CNF, such that $\phi \triangleq \phi_1 \wedge \dots \wedge \phi_n$ and each ϕ_i (a *partition*) is a conjunction of clauses. Given a refutation R of ϕ and a sequence of labeled interpolation systems $Itp_{L_1}, \dots, Itp_{L_n}$, we compute a sequence of interpolants I_1, \dots, I_n from R . Viewing ϕ as an unsatisfiable conjunction of the form $A \wedge B$, each I_i is obtained by setting ϕ_i to A and all the other ϕ_j to B ($I_i \triangleq Itp_{L_i}(\phi_i, \phi_1 \wedge \dots \wedge \phi_{i-1} \wedge \phi_{i+1} \wedge \dots \wedge \phi_n)$). These n ways of splitting the formula ϕ into A and B will be referred to as *configurations*. We prove that $I_1 \wedge \dots \wedge I_n \rightarrow \perp$ (requirement Req1), if for each i : $Itp_{L_i} \equiv Itp_P$ ³, and then generalize to any sequence of interpolation systems stronger than Itp_P .

3.2 Proof for Pudlák's System

Pudlák's system is symmetric, i.e., $Itp_P(\phi_1, \phi_2) = \neg Itp_P(\phi_2, \phi_1)$. Thus for $n = 2$, $I_1 = \neg I_2$; it follows that $I_1 \wedge I_2 \rightarrow \perp$. We will prove that Req1 holds for $n = 3$ and can be extended to an arbitrary number of partitions n . Table 3 shows the class and the color that a variable p assumes in the three configurations, depending on presence of p in the three partitions.

Lemma 1. *For Pudlák's interpolation system, $I_1 \wedge I_2 \wedge I_3 \rightarrow \perp$.*

² In [8], a notion of *symmetric interpolant* overlapping with Req1 is used. We avoid this name for its easy confusion with symmetry of interpolants, a known property of the interpolants generated by Itp_P .

³ Recall that Itp_P is applied to n distinct configurations, so it may be associated with different labelings for different values of i .

Table 3. Variables coloring in Itp_P for $n = 3$

p in ?	Variable class, color for each configuration			
	$A \triangleq \phi_1$, $B \triangleq \phi_2 \wedge \phi_3$	$A \triangleq \phi_2$, $B \triangleq \phi_1 \wedge \phi_3$	$A \triangleq \phi_3$, $B \triangleq \phi_1 \wedge \phi_2$	
ϕ_1	A, a	B, b	B, b	B, b
ϕ_2	B, b	A, a	A, a	B, b
ϕ_3	B, b	B, b	A, a	A, a
ϕ_1, ϕ_2	AB, ab	AB, ab	B, b	B, b
ϕ_1, ϕ_3	AB, ab	B, b	AB, ab	AB, ab
ϕ_2, ϕ_3	B, b	AB, ab	AB, ab	AB, ab
ϕ_1, ϕ_2, ϕ_3	AB, ab	AB, ab	AB, ab	AB, ab

Proof (by structural induction). We show that for any clause C in the refutation, the conjunction of its partial interpolants in the three configurations is unsatisfiable. In accordance with Tables 1, 2, we refer to the partial interpolants of the antecedents as I^+ and I^- with a subscript i to identify the corresponding configuration.

Base case (leaf). A clause C can belong either to ϕ_1 , ϕ_2 or ϕ_3 . In each case the clause belongs to A in one configuration and to B in the other two configurations; this implies that the conjunction of its partial interpolants contains one \perp element (see Table 2), which makes the conjunction unsatisfiable.

Inductive step (inner node). The inductive hypothesis (i.h.) consists of $I_1^+ \wedge I_2^+ \wedge I_3^+ \rightarrow \perp$, $I_1^- \wedge I_2^- \wedge I_3^- \rightarrow \perp$. A pivot p can either be local to a partition or shared by at least two partitions. If local, it has color a in one configuration and b in all the others; let us assume w.l.o.g. that p is local to ϕ_1 . In Itp_P the partial interpolants for the three configurations are $I_1^+ \vee I_1^-$, $I_2^+ \wedge I_2^-$, $I_3^+ \wedge I_3^-$:

$$(I_1^+ \vee I_1^-) \wedge I_2^+ \wedge I_2^- \wedge I_3^+ \wedge I_3^- \leftrightarrow \\ (I_1^+ \wedge I_2^+ \wedge I_2^- \wedge I_3^+ \wedge I_3^-) \vee (I_1^- \wedge I_2^+ \wedge I_2^- \wedge I_3^+ \wedge I_3^-) \xrightarrow{i.h.} \perp$$

If shared, p has color b in (at most) one configuration and ab in the other ones. Let us assume w.l.o.g. that p is shared between ϕ_2 and ϕ_3 . The three partial interpolants are $I_1^+ \wedge I_1^-$, $(I_2^+ \vee p) \wedge (I_2^- \vee \bar{p})$, $(I_3^+ \vee p) \wedge (I_3^- \vee \bar{p})$:

$$I_1^+ \wedge I_1^- \wedge (I_2^+ \vee p) \wedge (I_2^- \vee \bar{p}) \wedge (I_3^+ \vee p) \wedge (I_3^- \vee \bar{p}) \xrightarrow{\text{vintroduction}} \\ (I_1^+ \vee p) \wedge (I_1^- \vee \bar{p}) \wedge (I_2^+ \vee p) \wedge (I_2^- \vee \bar{p}) \wedge (I_3^+ \vee p) \wedge (I_3^- \vee \bar{p}) \leftrightarrow \\ (p \vee (I_1^+ \wedge I_2^+ \wedge I_3^+)) \wedge (\bar{p} \vee (I_1^- \wedge I_2^- \wedge I_3^-)) \xrightarrow{\text{resolution}} \\ (I_1^+ \wedge I_2^+ \wedge I_3^+) \vee (I_1^- \wedge I_2^- \wedge I_3^-) \xrightarrow{i.h.} \perp$$

□

Lemma 11 can be extended to an arbitrary number of partitions:

Theorem 1. For Pudlák's interpolation system, $I_1 \wedge \dots \wedge I_n \rightarrow \perp$.

Proof. Proof by structural induction as in Lemma 11.

Base case (leaf). As in the proof of Lemma 11 with n partitions instead of 3.

Inductive step (inner node). If the pivot p is local to a partition, the same argumentation of the proof of Lemma 11 holds. If p is shared, it might assume colors b (possibly in several configurations) or ab . Multiple applications of the \vee introduction rule and one resolution step yield the result. \square

3.3 Proof Generalization

Now we generalize Theorem 11 to a family of sequences of interpolation systems.

Theorem 2. *For any sequence of interpolation systems $Itp_{L_1}, \dots, Itp_{L_n}$, s.t. every Itp_{L_i} is stronger than Itp_P , $I_1 \wedge \dots \wedge I_n \rightarrow \perp$. (see Figure 2).*

Proof. Let \tilde{I}_i be $Itp_P(\phi_i, \phi_1 \wedge \dots \wedge \phi_{i-1} \wedge \phi_{i+1} \wedge \dots \wedge \phi_n)$; we have that $I_i \rightarrow \tilde{I}_i$ (recall the partial order on systems defined in §2.3). This implies $I_1 \wedge \dots \wedge I_n \rightarrow \tilde{I}_1 \wedge \dots \wedge \tilde{I}_n$, which in turn implies \perp . \square

The result does not necessary hold for systems weaker than Itp_P . For example, let us consider $Itp_{M'}$. A simple counterexample shows that $I_1 \wedge \dots \wedge I_n \rightarrow \perp$ does not hold even for a trivial formula with only two partitions; let $\phi_1 \triangleq (p \vee \bar{q}) \wedge r$, $\phi_2 \triangleq (\bar{p} \vee \bar{r}) \wedge q$:

$$\begin{array}{c} \text{Configuration } A \triangleq \phi_1, B \triangleq \phi_2: \\ \frac{\frac{p \vee \bar{q}[\perp] \quad \bar{p} \vee \bar{r}[p \wedge r]}{\bar{q} \vee \bar{r}[p \wedge r]} \quad r[\perp]}{\perp[(p \wedge r) \vee \bar{q}]} \end{array} \quad \begin{array}{c} \text{Configuration } A \triangleq \phi_2, B \triangleq \phi_1: \\ \frac{\frac{p \vee \bar{q}[\bar{p} \wedge q] \quad \bar{p} \vee \bar{r}[\perp]}{\bar{q} \vee \bar{r}[\bar{p} \wedge q]} \quad r[\bar{r}]}{\perp[(\bar{p} \wedge q) \vee \bar{r}]} \end{array}$$

Clearly, the interpolants $(p \wedge r) \vee \bar{q}$ and $(\bar{p} \wedge q) \vee \bar{r}$ are not mutually unsatisfiable: a partial model is \bar{q}, \bar{r} .

3.4 Application to Model Checking

We provide two examples of model checking algorithms, where the above setting occurs. In [8], the authors present an algorithm for iterative refinement of an abstraction of a transition relation. Initially, a coarse abstraction of the transition relation $\hat{T}_0 \equiv \text{true}$ is used. The abstraction \hat{T}_i is used to check reachability of error states represented by a predicate ψ from initial states represented by a predicate U . If error states are unreachable using the abstract transition relation, the system is safe. Otherwise, we have a trace from an initial state to an error state in \hat{T}_i of length n , for some n . Then, reachability of the error states in n steps is checked using the precise transition relation T . For this purpose, a precise

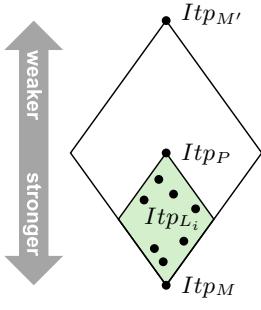


Fig. 2. The interpolation systems stronger than Itp_P (colored area)

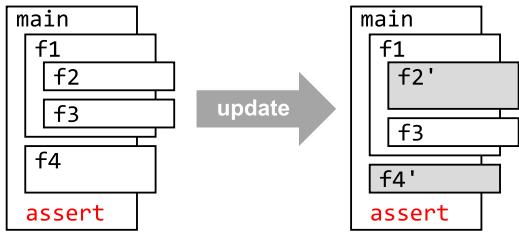


Fig. 3. Update check of a program with functions $f2$, $f4$ changed to $f2'$, $f4'$

bounded model checking formula is constructed and checked for satisfiability⁴:

$$U^{\langle 0 \rangle} \wedge T^{\langle 0 \rangle} \wedge T^{\langle 1 \rangle} \wedge \dots \wedge T^{\langle n-1 \rangle} \wedge \psi^{\langle n \rangle}$$

If satisfiable, a real error is found and the error trace is extracted from the satisfying assignment. If unsatisfiable, the corresponding interpolants are extracted from the refutation and used to strengthen the abstraction:

$$\hat{T}_{i+1} \triangleq \hat{T}_i \wedge I_1^{\langle 0 \rangle} \wedge I_2^{\langle -1 \rangle} \wedge \dots \wedge I_n^{\langle -n+1 \rangle}$$

The fact that $\hat{T}_{i+1} \rightarrow I_j^{\langle -j+1 \rangle}$, for $1 \leq j \leq n$, and the requirement Req1 yield:

$$U^{\langle 0 \rangle} \wedge \hat{T}_{i+1}^{\langle 0 \rangle} \wedge \hat{T}_{i+1}^{\langle 1 \rangle} \wedge \dots \wedge \hat{T}_{i+1}^{\langle n-1 \rangle} \wedge \psi^{\langle n \rangle} \rightarrow \perp$$

So the new transition relation \hat{T}_{i+1} does not contain any error trace of length n and it is a tighter abstraction than \hat{T}_i . For this reason, the algorithm terminates for finite state systems if Req1 holds. Otherwise, termination is not guaranteed.

Another example concerns software update checking. Figure 3 depicts a situation where a program is being updated. Under a suitable encoding (e.g., as in [17]), safety of the program (w.r.t. assertion violation) is equivalent to unsatisfiability of a formula of the form $\phi_{\text{main}} \wedge \phi_{f1} \wedge \phi_{f2} \wedge \phi_{f3} \wedge \phi_{f4}$, where each conjunct represents one of the functions `main`, `f1`, `f2`, `f3`, `f4`. If the original program is safe, the formula is unsatisfiable and we can generate interpolants I_{main} , I_{f1} , I_{f2} , I_{f3} , I_{f4} . The requirement Req1 yields $I_{\text{main}} \wedge I_{f1} \wedge I_{f2} \wedge I_{f3} \wedge I_{f4} \rightarrow \perp$ and thus also $\phi_{\text{main}} \wedge \phi_{f1} \wedge I_{f2} \wedge \phi_{f3} \wedge I_{f4} \rightarrow \perp$. Now, to prove safety of the updated program, it suffices to show that $\phi_{f2'} \rightarrow I_{f2}$ and $\phi_{f4'} \rightarrow I_{f4}$. In other words, that the abstractions I_{f2} and I_{f4} of functions `f2` and `f4` are still valid abstractions for the changed functions `f2'` and `f4'`. Note that this is a local and thus

⁴ In accordance with [8], we expect the transition relation to be a relation over state variables and their primed versions for the next state values and we use superscript $\langle i \rangle$ to indicate addition of i primes (or removal if i is negative).

computationally cheap check. Without the requirement Req1, the whole formula for the entire updated program would have to be constructed and checked again, which could require many more computational resources.

Theorem 2 offers the choice of interpolation systems generating interpolants of different strength satisfying Req1. In this second example, the benefit of a stronger interpolant is a tighter abstraction, i.e., the interpolant more closely reflects the actual behavior of the corresponding function. On the other hand, a weaker interpolant is more permissive. So it is more likely to remain a valid abstraction, when the corresponding function gets updated.

4 Inductive Sequence of Interpolants

This section analyzes the generation of a sequence of interpolants from the conjuncts of an unsatisfiable formula; the requirement (Req2) is to guarantee that the sequence is inductive [7]. We formally describe the problem, proving that the requirement is satisfied if the interpolants are produced using Pudlák's interpolation system, and later generalize the result to a particular family of systems in the lattice. We conclude by illustrating the applications to model checking.

4.1 Problem Description

As input, we assume an unsatisfiable formula ϕ in CNF, such that $\phi \triangleq \phi_1 \wedge \dots \wedge \phi_n$ and each ϕ_i is a conjunction of clauses. Given a refutation R of ϕ and a sequence of labeled interpolation systems $Itp_{L_0}, \dots, Itp_{L_n}$, we compute a sequence of interpolants I_0, I_1, \dots, I_n from R ; I_i is obtained by setting $\phi_1 \wedge \dots \wedge \phi_i$ to A and $\phi_{i+1} \wedge \dots \wedge \phi_n$ to B ($I_i \triangleq Itp_{L_i}(\phi_1 \wedge \dots \wedge \phi_i, \phi_{i+1} \wedge \dots \wedge \phi_n)$), in particular $I_0 \equiv Itp_{L_0}(\top, \phi) \equiv \top$ and $I_n \equiv Itp_{L_n}(\phi, \top) \equiv \perp$. These ways of splitting the formula ϕ into A and B will be referred to as *configurations*.

We prove that I_0, I_1, \dots, I_n is an inductive sequence of interpolants: for every i , $I_i \wedge \phi_{i+1} \rightarrow I_{i+1}$ holds (requirement Req2) if, for every i , $Itp_{L_i} \equiv Itp_P$ (as in the previous setting, Itp_P can be associated with different labelings for different values of i). Then we generalize to a family of sequences of interpolation systems.

Notice that, for a given i , only two configurations need to be taken into account, the first associated with I_i ($A \triangleq \phi_1 \wedge \dots \wedge \phi_i$, $B \triangleq \phi_{i+1} \wedge \dots \wedge \phi_n$), the second with I_{i+1} ($A \triangleq \phi_1 \wedge \dots \wedge \phi_{i+1}$, $B \triangleq \phi_{i+2} \wedge \dots \wedge \phi_n$); ϕ_{i+1} is the only subformula shared between A and B .

Since the proof is independent of i , to simplify the notation we will represent $\phi_1 \wedge \dots \wedge \phi_i$ as X , ϕ_{i+1} as S , $\phi_{i+2} \wedge \dots \wedge \phi_n$ as Y (so that the formula is $X \wedge S \wedge Y$), I_i as I , I_{i+1} as J and $I_i \wedge \phi_{i+1} \rightarrow I_{i+1}$ as $I \wedge S \rightarrow J$.

4.2 Proof for Pudlák's System

Theorem 3. *For Pudlák's interpolation system, $I \wedge S \rightarrow J$.*

Table 4. Variables coloring for Definition II

p in ?	Variable class, color for each configuration	
	$A \triangleq X, B \triangleq S \wedge Y$	$A \triangleq X \wedge S, B \triangleq Y$
X	A, a	A, a
S	A, a	B, b
Y	B, b	B, b
X, S	$AB, \alpha \in \{a, b, ab\}$	A, a
S, Y	B, b	$AB, \beta \in \{a, b, ab\}$
X, Y	$AB, \gamma_1 \in \{a, b, ab\}$	$AB, \gamma_2 \in \{a, b, ab\}$
X, S, Y	$AB, \delta_1 \in \{a, b, ab\}$	$AB, \delta_2 \in \{a, b, ab\}$

Proof. By the above definitions, $I \equiv Itp(X, S \wedge Y)$ and $J \equiv Itp(X \wedge S, Y) = \neg Itp(Y, X \wedge S)$ (by symmetry of Pudlák's system). Denoting $K \triangleq Itp(S, X \wedge Y)$, Lemma II states that $I \wedge K \wedge \neg J \rightarrow \perp$. Since $S \rightarrow K$, then $I \wedge S \wedge \neg J \rightarrow \perp$, that is $I \wedge S \rightarrow J$. \square

4.3 Proof Generalization

We will now prove that $I \wedge S \rightarrow J$ holds in all the sequences of interpolation systems that comply with particular coloring restrictions. As shown in Table II, two configurations are to be considered, which share the conjunct S . By $C|_{1,\sigma}$ and $C|_{2,\sigma}$ we denote the restriction of a clause C to the literals of color σ according to the labeling of configurations 1 and 2, respectively.

To simplify the proofs we initially enforce a set of constraints, so that the color taken by the occurrence of a variable in a clause in the two configurations is consistent; we will later show that the result still holds if the constraints are relaxed.

Definition 1 (Coloring constraints). We define a set of coloring constraints (CC) over Table II as follows: $\alpha = a$, $\beta = b$, $\gamma_1 = \gamma_2$, $\delta_1 = \delta_2$.

Lemma 2. $I \wedge S \rightarrow J$, assuming the coloring constraints of Definition II

Proof (by structural induction). We prove that, for any clause C in a refutation of $X \wedge S \wedge Y$, $f_C \wedge I(C) \wedge S \wedge \neg J(C) \rightarrow \perp$, where f_C is an additional constraint (to be determined), dependent on C , that becomes empty at the end of the proof ($f_\perp \equiv \top$). For simplicity we drop the parameter C in I , J . I^+ , I^- are defined as in the previous setting, similarly J^+ and J^- .

Base case (leaf). Case splitting on C (refer to Table II):

$$\begin{aligned} C \in X : & \quad I \equiv C|_{1,b} \text{ and } J \equiv C|_{2,b} \\ C \in S : & \quad I \equiv \neg(C|_{1,a}) \text{ and } J \equiv C|_{2,b} \\ C \in Y : & \quad I \equiv \neg(C|_{1,a}) \text{ and } J \equiv \neg(C|_{2,a}) \end{aligned}$$

We construct f_C in order to simultaneously satisfy the following conditions:

$$C \in X : \quad f_C \wedge C|_{1,b} \wedge S \wedge \neg(C|_{2,b}) \rightarrow \perp$$

$$\begin{aligned} C \in S : & f_C \wedge \neg(C|_{1,a}) \wedge S \wedge \neg(C|_{2,b}) \rightarrow \perp \\ C \in Y : & f_C \wedge \neg(C|_{1,a}) \wedge S \wedge C|_{2,a} \rightarrow \perp \end{aligned}$$

The CC constraints yield $C|_{1,b} \wedge \neg(C|_{2,b}) \rightarrow \perp$ and $\neg(C|_{1,a}) \wedge C|_{2,a} \rightarrow \perp$; as for $\neg(C|_{1,a}) \wedge \neg(C|_{2,b})$, it “counteracts” the literals of S with variables in X, S and S, Y and X, S, Y (colored a or b). The literals left are those whose variables are in S (denoted by $C|_S$) and those in X, S, Y colored ab (denoted by $C|_{ab}^{XSY}$); it is thus sufficient to set $f_C \triangleq \neg(C|_S \vee C|_{ab}^{XSY})$.

Inductive step (inner node). The inductive hypothesis (i.h.) provides that $f_{(C+ \vee p)} \wedge I^+ \wedge S \wedge \neg J^+ \rightarrow \perp$ and $f_{(C- \vee \bar{p})} \wedge I^- \wedge S \wedge \neg J^- \rightarrow \perp$.

Now $f_C \equiv f_{(C+ \vee C^-)}$, that is $\neg((C^+ \vee C^-)|_S) \wedge \neg((C^+ \vee C^-)|_{ab}^{XSY})$.

We have:

$$f_C \rightarrow f_{C^+} \wedge f_{C^-} \quad (1)$$

since $\neg((C^+ \vee C^-)|_S) \leftrightarrow \neg(C^+|_S) \wedge \neg(C^-|_S)$ and $(C^+|_{ab}^{XSY}) \vee (C^-|_{ab}^{XSY}) \rightarrow ((C^+ \vee C^-)|_{ab}^{XSY})$ ⁵.

Case splitting based on the presence of the pivot p in $X/S/Y$ (see Table 4):

Case 1 (p in X).

$$\begin{aligned} & f_C \wedge S \wedge I \wedge \neg J \leftrightarrow \\ & f_C \wedge S \wedge (I^+ \vee I^-) \wedge \neg(J^+ \vee J^-) \leftrightarrow \\ & f_C \wedge S \wedge (I^+ \vee I^-) \wedge \neg J^+ \wedge \neg J^- \leftrightarrow \\ & (f_C \wedge S \wedge I^+ \wedge \neg J^+ \wedge \neg J^-) \vee (f_C \wedge S \wedge I^- \wedge \neg J^- \wedge \neg J^+) \rightarrow \\ & \quad (f_C \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_C \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{(1)} \\ & \quad (f_{C^+} \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_{C^-} \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{(2)} \\ & \quad (f_{(C+ \vee p)} \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_{(C- \vee \bar{p})} \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{\text{i.h.}} \perp \end{aligned}$$

where (2) holds since p is restricted.

Case 2 (p in S).

$$\begin{aligned} & f_C \wedge S \wedge I \wedge \neg J \leftrightarrow \\ & f_C \wedge S \wedge (I^+ \wedge I^-) \wedge \neg(J^+ \vee J^-) \leftrightarrow \\ & f_C \wedge S \wedge I^+ \wedge I^- \wedge \neg J^+ \wedge \neg J^- \leftrightarrow \\ & (f_C \wedge S \wedge I^+ \wedge \neg J^+) \wedge (f_C \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{(1)} \\ & (f_{C^+} \wedge S \wedge I^+ \wedge \neg J^+) \wedge (f_{C^-} \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{(2)} \\ & (f_{C^+} \wedge S \wedge I^+ \wedge \neg J^+ \wedge \bar{p}) \vee (f_{C^-} \wedge S \wedge I^- \wedge \neg J^- \wedge p) \leftrightarrow^{(3)} \\ & (f_{(C+ \vee p)} \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_{(C- \vee \bar{p})} \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{\text{i.h.}} \perp \end{aligned}$$

⁵ Notice that the inverse of the last implication does not hold; in fact, a variable in X, S, Y could have, e.g., color ab in $C^+ \vee C^-$ because it has color a in C^+ and color b in C^- (recall the definition of \sqcup in §2.3), which means that it appears in $(C^+ \vee C^-)|_{ab}^{XSY}$ but gets restricted both in $C^+|_{ab}^{XSY}$ and in $C^-|_{ab}^{XSY}$.

where (2) holds since $\phi \wedge \chi \rightarrow (\phi \wedge p) \vee (\chi \wedge \bar{p})$, (3) since $f_{C_i} \wedge \bar{p} \leftrightarrow f_{(C_i \vee p)}$.

Case 3 (p in Y). Dual to Case 1.

Case 4 (p in X, S). As for Case 1.

Case 5 (p in S, Y). As for Case 3.

Case 6 (p in X, Y). As for Case 1 or Case 3 if color is either a or b . If ab :

$$\begin{aligned}
& f_C \wedge S \wedge I \wedge \neg J \leftrightarrow \\
& f_C \wedge S \wedge ((I^+ \vee p) \wedge (I^- \vee \bar{p})) \wedge \neg((J^+ \vee p) \wedge (J^- \vee \bar{p})) \leftrightarrow \\
& f_C \wedge S \wedge (I^+ \vee p) \wedge (I^- \vee \bar{p}) \wedge (\neg(J^+ \vee p) \vee \neg(J^- \vee \bar{p})) \rightarrow \\
& (f_C \wedge S \wedge (I^+ \vee p) \wedge \neg(J^+ \vee p)) \vee (f_C \wedge S \wedge (I^- \vee \bar{p}) \wedge \neg(J^- \vee \bar{p})) \leftrightarrow \\
& (f_C \wedge S \wedge (I^+ \vee p) \wedge \neg J^+ \wedge \bar{p}) \vee (f_C \wedge S \wedge (I^- \vee \bar{p}) \wedge \neg J^- \wedge p) \rightarrow \\
& (f_C \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_C \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{(1)} \\
& (f_{C^+} \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_{C^-} \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{(2)} \\
& (f_{(C \vee p)} \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_{(C \vee \bar{p})} \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{\text{i.h.}} \perp
\end{aligned}$$

where (2) holds since p is restricted.

Case 7 (p in X, S, Y). As for Case 1 or Case 3 if color is either a or b , since p is restricted. As for Case 6 if color is ab , but last three lines are replaced by:

$$\begin{aligned}
& (f_C \wedge S \wedge (I^+ \vee p) \wedge \neg J^+ \wedge \bar{p} \wedge \bar{p}) \vee (f_C \wedge S \wedge (I^- \vee \bar{p}) \wedge \neg J^- \wedge p \wedge p) \rightarrow \\
& (f_C \wedge S \wedge I^+ \wedge \neg J^+ \wedge \bar{p}) \vee (f_C \wedge S \wedge I^- \wedge \neg J^- \wedge p) \rightarrow^{(1)} \\
& (f_{C^+} \wedge S \wedge I^+ \wedge \neg J^+ \wedge \bar{p}) \vee (f_{C^-} \wedge S \wedge I^- \wedge \neg J^- \wedge p) \rightarrow^{(2)} \\
& (f_{(C \vee p)} \wedge S \wedge I^+ \wedge \neg J^+) \vee (f_{(C \vee \bar{p})} \wedge S \wedge I^- \wedge \neg J^-) \rightarrow^{\text{i.h.}} \perp
\end{aligned}$$

where (2) holds since $f_{C_i} \wedge \bar{p} \leftrightarrow f_{(C_i \vee p)}$. \square

Lemma 3. *The CC constraints of Definition 1 can be relaxed as follows: $\alpha \preceq a$, $b \preceq \beta$, $\gamma_1 \preceq \gamma_2$, $\delta_1 \preceq \delta_2$.*

Proof. Let I, J be interpolants generated using interpolation systems according to the constraints CC in Def. 1. Let us use primed variables to represent the relaxed constraints of Lemma 3 as $\alpha' \preceq \alpha = a$ (i.e., any α'), $b = \beta \preceq \beta'$ (i.e., any β'), $\gamma'_1 \preceq \gamma_1 = \gamma_2 \preceq \gamma'_2$, $\delta'_1 \preceq \delta_1 = \delta_2 \preceq \delta'_2$ and let I', J' represent the corresponding interpolants. Recalling the order $b \preceq ab \preceq a$ over colors, which induces a partial order over interpolation systems (see §2.3), we get $I' \rightarrow I$ and $J \rightarrow J'$. Thus, the relaxed constraints yield: $I' \wedge S \wedge \neg J' \rightarrow I \wedge S \wedge \neg J \rightarrow \perp$. \square

The above considerations lead to the following result:

Theorem 4. *For any sequence of interpolation systems $\text{It}_{L_0}, \dots, \text{It}_{L_n}$, which respects the relaxed constraints of Lemma 3, I_0, I_1, \dots, I_n is an inductive sequence of interpolants, i.e., $I_i \wedge \phi_{i+1} \rightarrow I_{i+1}$, for $0 \leq i < n$.*

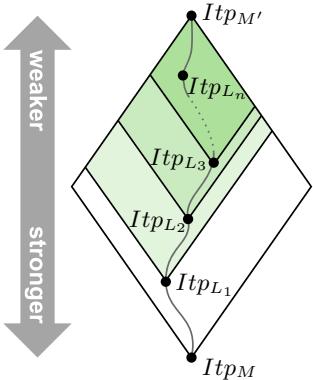


Fig. 4. Weakening interpolation systems

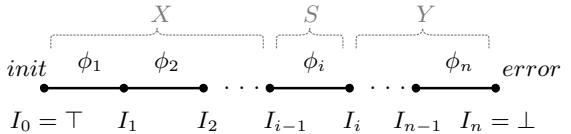


Fig. 5. Spurious error trace annotated by interpolants

4.4 Application to Model Checking

The above setting occurs, for example, during counterexample-guided abstraction refinement. Given a spurious error trace, the goal is to annotate nodes of the abstract reachability tree with an inductive sequence of formulae that together rule out the trace. The trace is represented as a formula $\phi_1 \wedge \dots \wedge \phi_n$, which is constructed from the SSA form of the trace and which is unsatisfiable if and only if the error trace is infeasible. If so, a sequence of interpolants I_i is created by means of a sequence of interpolation systems as $I_i \triangleq Itp_{L_i}(\phi_1 \wedge \dots \wedge \phi_i, \phi_{i+1} \wedge \dots \wedge \phi_n)$, $I_0 \equiv \top$, and $I_n \equiv \perp$, as depicted in Fig. 5 (along with the partitioning of the path formula into X , S , and Y for the purposes of Lemma 2). In addition, Req2 requires the sequence of interpolants to be inductive, i.e., $I_i \wedge \phi_{i+1} \rightarrow I_{i+1}$. In which case, the error trace is removed from the refined abstraction. However, such a sequence of interpolants is not inductive in general and thus the same error trace may remain also in the refined abstraction, should Req2 be violated. As already mentioned, refinement phases of tools like BLAST [1], IMPACT [13] and WOLVERINE [9], as well as some BMC techniques [18] rely on this fact.

Theorem 4 ensures that Req2 is satisfied by interpolants derived using interpolation systems weakening towards the end of the error trace (depicted in Fig. 4). It is thus possible to choose an interpolation system depending on the instruction at the current position along the error trace (i.e., the current S in the language of Lemma 2). As an example, some instructions in the error trace may trigger generation of weaker interpolants (i.e., a more coarse abstraction). In practice, this would affect the speed of convergence of the refinement loop.

5 Related Work

In model checking, interpolation is a common means for abstraction. Interpolation is used as an abstract post-image operator in hardware bounded model

checking [10], the interpolant is generated from the proof of unsatisfiability of a bounded model checking formula so that it represents a superset of states reachable from the initial states. Faster convergence of the model checking algorithm applying this method of abstraction is observed during experiments. Interpolation is also used in concolic execution to propagate reasons of trace infeasibility backward towards the start of the program [14]. This allows discarding infeasible traces as early as possible and thus saving the effort of evaluating them. In software bounded model checking, function summaries can be created using interpolation [17]; these are employed during analysis of different properties to represent a function body without the need to process its whole call tree. Interpolation also proves to be very useful in refining predicate abstraction based on spurious counterexamples [7]. Here, interpolation is used to derive new predicates that rule out the spurious error traces. The listed works describe applications of interpolation in model checking; see [12] for a comprehensive list. Typically, the authors limit themselves to either Pudlák's or McMillan's algorithms without considering further variation in the strength of interpolants. We believe that all these techniques would benefit from choosing among interpolants of appropriate strengths. The results of this paper provide safe boundaries for such a choice.

Other related work concerns the actual generation of interpolants. Pudlák [15] shows that interpolants can be derived in linear time with respect to the given refutation. McMillan [11] proposes a different algorithm that produces logically stronger interpolants and addresses both propositional logic and a quantifier free combination of the theories of uninterpreted functions and linear arithmetic. In [2], local proof transformations are presented that (by reordering proofs and removing so called ab-mixed predicates) can change a refutation produced by a standard SMT-solver so that it becomes suitable for interpolant generation. Authors of [5] provide a generalized algorithm for interpolation that subsumes both Pudlák's and McMillan's algorithms. They also show that a complete lattice of interpolants ordered by the implication relation can be systematically derived from a given refutation. However, they do not study the limits with regard to the actual application of interpolants of differing strength in model checking. Building upon [5], our work provides this missing connection and defines and proves these boundaries in the particular model checking settings.

In this paper, we consider two classes of model checking approaches that put additional requirements on the resulting interpolants. These requirements were previously formulated in the literature (Req1 in [8] and Req2 in [7]). Until now, however, the conditions under which they hold have not been thoroughly studied, in particular in the context of different interpolation systems of [5]. Novelty of our work lies in the fact that we provide constraints on the complete lattice of interpolants that, when obeyed, ensure satisfaction of the requirements.

6 Conclusion

Interpolants are not unique and may vary in strength. The effects of using interpolants of different strength in model checking can be substantial and are

yet to be properly studied. However, common applications of interpolation in model checking put additional requirements that (as we show) are not satisfied in general, specifically when interpolants of various strengths are generated by different interpolation systems. In this paper, for two classes of model checking techniques employing interpolation, we showed the safe boundaries for varying the strength of interpolants, proving the limitations under which the requirements are satisfied. Our theoretical result enables study of the effects of the interpolants strength on the model checking algorithms. Since our result is not limited to an ad-hoc proof system, any state-of-the-art solver can be used to generate proofs used for interpolation. Strength and size of interpolants can be also affected by proof manipulation procedures as shown in [16]. We intend to address the above questions in our future work.

References

1. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker Blast: Applications to Software Engineering. *Int. J. STTT* 9, 505–525 (2007)
2. Bruttomesso, R., Rollini, S.F., Sharygina, N., Tsitovich, A.: Flexible Interpolation with Local Proof Transformations. In: ICCAD 2010, pp. 770–777. IEEE (2010)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
4. Craig, W.: Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. of Symbolic Logic*, 269–285 (1957)
5. D’Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant Strength. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010)
6. Heizmann, M., Hoenicke, J., Podelski, A.: Nested Interpolants. In: POPL 2010, pp. 471–482. ACM (2010)
7. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: POPL 2004, pp. 232–244. ACM (2004)
8. Jhala, R., McMillan, K.L.: Interpolant-Based Transition Relation Approximation. *Logical Methods in Computer Science* 3(4) (2007)
9. Kroening, D., Weissenbacher, G.: Interpolation-Based Software Verification with WOLVERINE. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)
10. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
11. McMillan, K.L.: An Interpolating Theorem Prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)
12. McMillan, K.L.: Applications of Craig Interpolants in Model Checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)
13. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
14. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)

15. Pudlák, P.: Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *Journal of Symbolic Logic* 62(3), 981–998 (1997)
16. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An Efficient and Flexible Approach to Resolution Proof Reduction. In: Barner, S., Kroening, D., Raz, O. (eds.) HVC 2010. LNCS, vol. 6504, pp. 182–196. Springer, Heidelberg (2010)
17. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based Function Summaries in Bounded Model Checking. In: HVC 2011. LNCS (2011) (to appear)
18. Vizel, Y., Grumberg, O.: Interpolation-sequence Based Model Checking. In: FM-CAD 2009, pp. 1–8. IEEE (2009)

Detecting Fair Non-termination in Multithreaded Programs^{*}

Mohamed Faouzi Atig¹, Ahmed Bouajjani², Michael Emmi^{2, **}, and Akash Lal³

¹ Uppsala University, Sweden

mohamed_faouzi.atig@it.uu.se

² LIAFA, Université Paris Diderot, France

{abou, mje}@liafa.jussieu.fr

³ Microsoft Research, Bangalore, India

akashl@microsoft.com

Abstract. We develop compositional analysis algorithms for detecting non-termination in multithreaded programs. Our analysis explores fair and ultimately-periodic executions—i.e., those in which the infinitely-often enabled threads repeatedly execute the same sequences of actions over and over. By limiting the number of context-switches each thread is allowed along any repeating action sequence, our algorithm quickly discovers practically-arising non-terminating executions. Limiting the number of context-switches in each period leads to a compositional analysis in which we consider each thread separately, in isolation, and reduces the search for fair ultimately-periodic executions in multithreaded programs to state-reachability in sequential programs. We implement our analysis by a systematic code-to-code translation from multithreaded programs to sequential programs. By leveraging standard sequential analysis tools, our prototype tool MUTANT is able to discover fair non-terminating executions in typical mutual exclusion protocols and concurrent data-structure algorithms.

1 Introduction

Multithreaded programming is the predominant style for implementing parallel and reactive single-processor software. A multithreaded program is composed of several sequentially-executing *threads* who share the same memory address space. As a thread’s operations on shared memory generally do not commute with the operations of others, each *schedule*—i.e., each distinct order on the actions of different threads—leads to distinct program behavior. Generally speaking, the schedule of inter-thread execution relies on factors external to the program, such as processor utilization and I/O activity. Though some programming errors are witnessed in many different schedules, and are thus likely to be discovered by testing, others manifest only in a small number of rarely-encountered schedules; these *Heisenbugs* are notoriously difficult to debug.

The correctness criteria for multithreaded programs generally include both safety and liveness conditions, and ensuring safety can threaten liveness. For instance, to ensure linearizability—i.e., the result of concurrently executing operations is equivalent to

* Partially supported by the project ANR-09-SEGI-016 Veridyc.

** Supported by a Fondation Sciences Mathématiques de Paris post-doctoral fellowship.

some sequential execution of the same operations—concurrent data structure implementations often employ a *retrying* mechanism [9] (see Figure 1c for a simple instance): a validation phase before the effectuation of each operation ensures concurrent modifications have not interfered; when validation fails, the operation is simply attempted again. A priori nothing prevents an operation from being retried forever. Retry is also a mechanism used in mutual exclusion protocols. For instance, a common solution to the dining philosophers problem proposes that philosophers drop the fork they first picked up when they cannot obtain the second fork—presumably because a neighboring philosopher already holds the second. Though this scheme avoids deadlock, it also leads to non-terminating executions in which no philosophers ever eat; particularly when each philosopher picks up his first fork, finds his neighbor has the other, and then all release their first fork, repeatedly; Figure 1b illustrates a simplification of this pattern. As such retrying raises the possibility that some or all interfering operations are never completed even under *fair* schedules—repeatedly failing operations already execute infinitely often—one does want to ensure that concurrent operations do always terminate. Note that unlike in sequential programs, where interesting non-terminating executions involve ever diverging data values, non-terminating executions in multithreaded programs also involve repeated inter-thread interference, even over small finite data domains (see Figure 1).

Proving the absence of programming errors such as assertion violations, and unintentional non-termination due to inter-thread interference, in multithreaded programs is difficult precisely because of the enormous number of possible schedules which need be considered. Automated approaches based on model checking are highly complex—e.g., computing state-reachability is PSPACE-complete when threads are finite state [10], and undecidable when threads are recursive [23]—and are susceptible to state-explosion; naïve approaches are unlikely to scale to realistic programs. Otherwise, modular deductive verification techniques may apply, though they require programmer-supplied invariants, which for multithreaded programs are regarded as difficult to divine. Furthermore, a failed verification attempt may only prove that the supplied invariants are insufficient, rather than the existence of a programming error.

Instead of exhaustive program exploration, recent approaches to detecting safety violations (e.g., assertion violations) have focused on exploring only a representative subset of program behaviors by limiting inter-thread interaction [22, 21, 17, 15, 3]; for instance, Qadeer and Rehof [21] consider only executions with a given number $k \in \mathbb{N}$ of *context switches* between threads. Though techniques like *context-bounding* are clearly incomplete for any given $k \in \mathbb{N}$, every execution is considered in the limit as k approaches infinity, and small values of k have proved to provide great coverage [17] and uncover subtle bugs [13] in practice. The bounded analysis approach is particularly attractive since it enables compositional reasoning: each thread can be considered separately, in isolation, once the number of environmental interactions is fixed. This fact has been exploited by the so-called “sequentializations” which reduce multithreaded state-reachability under an interaction bound to state-reachability in a polynomially-sized sequential program [15, 11, 7, 3], leading to efficient analyses. Conveniently these reductions allow leveraging highly-developed sequential program analysis tools for multithreaded program analysis.

Fig. 1. Three programs with non-terminating executions. (a) Though the first thread may execute forever if the second never sets g to `true`, no such execution is fair. (b) Two threads repeatedly trying to validate their set values of g will keep retrying forever under a schedule which schedules each loop head just after the opposing thread's assignment. (c) As long as the first thread executes an iteration between each of the second thread's reads and validations of g , the second thread is never able to finish its operation.

Though these techniques seem promising for the detection of safety violations, they have been deemed inapplicable for detecting *liveness* violations, since, for instance, in any context-bounded execution, only one thread can execute infinitely often; interesting concurrency bugs such as unintentional yet coordinated non-termination require the participation of multiple infinitely-often executing threads. This limitation has effectively prevented the application of compositional bounded analyses to detecting liveness violations in multithreaded programs.

In this work we demonstrate that restricting thread interaction also leads to an effective technique for detecting liveness violations in recursive multithreaded programs—in particular we detect the presence of fair non-terminating executions. Though in general the problem of detecting non-terminating executions is very difficult, we restrict our attention to the simpler (recursively-enumerable yet still undecidable) case of fair *ultimately periodic* executions, which after a finite execution prefix (called the *stem*) ultimately repeat the same sequence of actions (the *lasso*) over and over again. Many interesting non-terminating executions occurring in practice are ultimately periodic. For instance, in the program of Figure 1b, every non-terminating execution must repeat the same sequence of statements on Lines 7, 8, 12, and 13. Similarly, every fair non-terminating execution of the program in Figure 1c must repeat the statements of Lines 5, 8, 12, 15, and 20–21. Thus focusing on periodically repeating executions is already quite interesting. Furthermore, every ultimately periodic execution is described with a finite number of thread contexts: those occurring during the stem, and those occurring during each iteration of the lasso; e.g., the non-terminating executions of each program in Figure 1 require just two contexts per thread: one per stem, and one per lasso.

By bounding the number of thread contexts we detect ultimately periodic executions compositionally, without exposing the local configurations of each thread to one another. We actually detect ultimately periodic executions that repeatedly encounter, along some lasso, the same sequence of shared global state valuations at thread context-switch points. Clearly ultimate state-repeatability is a sufficient condition for ultimate periodicity. We prove that this condition is necessary when the domain of shared global state valuations is finite. (This is not trivial in the presence of recursion, where threads access unbounded procedure stacks). Then, supposing each thread executes within k_1 contexts during the stem, and within k_2 contexts during each iteration of this lasso, its execution is summarized by an *interface* of $k = 2(k_1 + k_2)$ valuations $g_1g'_1 \dots g_kg'_k$: the shared global state valuations g_i and g'_i , resp., encountered at the beginning and end of each execution context during the stem and lasso. Given the possible bounded interfaces of each thread, we infer the existence of ultimately periodic executions by composing thread interfaces. Essentially, two context summaries $g_1g'_1$ and $g_2g'_2$ compose when $g'_1 = g_2$; by composing interfaces so that the valuation reached in the last context of the lasso match both the valuation reached in the last context of the stem, and the starting valuation of the first context of the lasso, we deduce the existence of a periodic computation.

We thus reduce the problem of detecting ultimately periodic computations to that of computing thread interfaces. Essentially, we must establish two conditions on an interface $g_1g'_1 \dots g_kg'_k$ of a thread t : first, the interface describes a valid thread computation, i.e., beginning from g_1 , t executing alone reaches g'_1 , and when resumed from the valuation g_2 , t executing alone reaches g'_2 , etc. Second, the interface is *repeatable*, i.e., each time t returns to its first lasso context i , t can again repeat the same sequence of global valuations $g_i g'_i \dots g_k g'_k$. Though both conditions reduce to (repeated) state-reachability for non-recursive programs, ensuring repeatability in recursive programs requires establishing equivalence of an unbounded number of procedure frames visited along each period of the lasso. An execution in which the procedure stack incurs a net decrease, for instance, along the lasso is not repeatable. We avoid explicitly comparing stack frames simply by noticing that along each period of any repeating execution there exists a procedure *keyframe* which is never returned from. By checking whether one keyframe can reach the same keyframe—perhaps with the first keyframe below on the procedure stack—in the same context number one period later, we ensure repeatability.

Finally, to ensure that the detected non-terminating executions are fair, we expose a bounded amount of additional information across thread interfaces. For the case of strong fairness, we observe that any thread t which does not execute during the lasso must be *blocked*, i.e., waiting on a synchronization object x which has not been signaled. Furthermore, in any fair execution, no concurrently executing thread may signal x , since otherwise t would become temporarily enabled—thus a violation of strong fairness. In this way, by ensuring thread interfaces agree on the set X of indefinitely waited-on synchronization objects, each thread can locally ensure no $x \in X$ is signaled during the lasso, and only threads waiting on some $x \in X$ are exempt from participating in the lasso.

As is the case for finding safety violations, the compositional fair non-termination analysis we describe in Section 3 has a convenient encoding as sequential program

analysis. In Section 4 we describe a code-to-code translation from multithreaded programs to sequential programs which violate an assertion exactly when the source program has a fair ultimately periodic execution with given bounds k_1 and k_2 on the number of stem and lasso contexts.¹ In Section 5 we discuss our implementation MUTANT, which systematically detects fair non-terminating executions in typical concurrent data structure and mutual exclusion algorithms.

2 Recursive Multithreaded Programs

We consider a simple but general multithreaded program model in which each of a statically-determined collection Tids of threads concurrently execute as recursive sequential programs which access a shared global state. For simplicity we suppose each program declares a single shared global variable g with domain Vals , and each procedure from a finite set Procs declares only a single parameter 1 , also of domain Vals ; furthermore each program statement is uniquely labeled from a set Locs of program locations. A (*procedure*) *frame* $f = \langle \ell, v \rangle$ is a program location $\ell \in \text{Locs}$ along with a local variable valuation $v \in \text{Vals}$, and a *configuration* $c = \langle g, \sigma \rangle$ is a shared global state valuation $g \in \text{Vals}$ along with a *local state map* $\sigma : \text{Tids} \rightarrow (\text{Locs} \times \text{Vals})^+$ mapping each thread t to a procedure frame stack $\sigma(t)$. The transition relation $\xrightarrow{t, \ell}$ between configurations is labelled by the active program location $\ell \in \text{Locs}$ and acting thread $t \in \text{Tids}$. We suppose a standard set of inter-procedural program statements, including assignment $x := e$, branching **if** e **then** s_1 **else** s_2 , and looping **while** e **do** s statements, lock **acquire** e and **release** e , and procedure **call** $x := p e$ and **return** e , where e are expressions from an unspecified grammar, s are labeled sub-statements, and $p \in \text{Procs}$. The definition of the transition relation is standard, as are the following:

Trace, Reachable: A *trace* π of a program P from a configuration c is a possibly empty transition-label sequence $a_0a_1a_2\dots$ for which there exists a configuration sequence $c_0c_1c_2\dots$ such that $c_0 = c$ and $c_j \xrightarrow{a_j} P c_{j+1}$ for all $0 \leq j < |\pi|$; each configuration $c_j = \langle g, \sigma \rangle$ (alternatively, the shared global valuation g) is said to be *reachable* from c by the finite trace $\pi_j = a_0a_1\dots a_{j-1}$.

Context: A *context of thread* t is a trace $\pi = a_0a_1\dots$ in which for all $0 \leq j < |\pi|$ there exists $\ell \in \text{Locs}$ such that $a_j = \langle t, \ell \rangle$; every trace is a context-sequence concatenation.

Enabled, Blocked, Fair: A thread $t \in \text{Tids}$ is *enabled* after a finite trace π if and only if there exists an a labeling a t -transition such that $\pi \cdot a$ is also a trace; otherwise t is *blocked*. An infinite trace is *strongly fair* (resp., *weakly fair*) if each infinitely-often (resp., continuously) enabled thread makes a transition infinitely often.

Checking typical safety and liveness specifications often reduces to finding whether certain program configurations are reachable, or determining whether fair infinite traces are possible. The following two problems are thus fundamental.

Problem 1 (State-Reachability). Given a configuration c of a program P , and a shared global state valuation g , is g reachable from c in P ?

¹ Technically, our reduction considers round-robin schedules of thread contexts.

Problem 2 (Fair Non-Termination). Given a configuration c of a program P , does there exist an infinite strongly (resp., weakly) fair trace of P from c ?

Even for recursive multithreaded programs accessing finite data, both problems are undecidable [23]. However, while state-reachability is recursively enumerable by examining all possible concurrent traces in increasing length, detecting non-terminating traces is more complex; from Yen [24] one deduces that the problem is not even semi-decidable. A simpler problem is to detect non-terminating traces which eventually repeat the same sequence of actions indefinitely. Formally, an infinite trace π is *ultimately periodic* when there exists two finite traces μ and v , called resp., the *stem* and *lasso*, such that $\pi = \mu \cdot v^\omega$. Then a key question is the detection of ultimately periodic traces.

Problem 3 (Fair Periodic Non-Termination). Given a configuration c of a program P , does there exist an ultimately periodic strongly (resp., weakly) fair trace of P from c ?

Periodic non-termination is also undecidable, yet still recursively enumerable—by examining all possible stems and lassos in increasing length. This implies that not all non-terminating executions are ultimately periodic. In principle, coordinating threads can construct phased executions in which each phase consists of an increasingly-longer sequence of actions, using their unbounded procedure stacks to simulate unbounded integer counters. Still, it is unclear whether non-periodic executions arise in practice. Our goal is to efficiently detect ultimately periodic fair traces where they exist.

3 Bounded Compositional Non-termination Analysis

Rather than incrementally searching for non-terminating executions by bounding the length of the considered stems and lassos, our discovery strategy bounds the number of thread contexts in the considered stems and lassos; this strategy is justified by the hypothesis that many interesting bugs are likely to occur within few contexts per thread [21, 17]. Notice, for instance, that the non-terminating executions of each of the programs in Figure 1 require only one context-switch per thread during their repeating sequences of actions. Formally for $k \in \mathbb{N}$, we say a trace $\pi = a_0 a_1 \dots$ is k *context-bounded* when there exist $j_1, j_2, \dots, j_k \in \mathbb{N}$ and $j_{k+1} = |\pi|$ such that $\pi = \pi_1 \pi_2 \dots \pi_k$ is a sequence of k thread contexts $\pi_i = a_{j_i} \dots a_{j_{i+1}-1}$; we refer to each j_i as a *context-switch point*. Though we expect many ultimately periodic traces to exhibit few context switches per period, context-bounding is anyhow complete in the limit as context-bounds approach infinity.

Remark 1. For every ultimately periodic trace $\mu \cdot v^\omega$ there exists $k_1, k_2 \in \mathbb{N}$ such that μ and v are, resp., k_1 and k_2 context-bounded.

In what follows, we show that for given stem and lasso context-bounds, resp., $k_1 \in \mathbb{N}$ and $k_2 \in \mathbb{N}$, the fair periodic non-termination problem reduces to the state-reachability problem in sequential programs; the salient feature of this reduction is compositionality: the resulting sequential program considers each thread independently, without explicitly representing thread-product states. The general idea is to show that all ultimately periodic executions can be decomposed into stem and lasso such that during each period of

the given lasso, each thread reencounters the same global valuations at context-switch points, and reencounters the same topmost stack-frame valuation; since each procedure stack must be non-decreasing over each lasso iteration, there must be some frame during each period which is never returned from. We show that detecting these repeated global valuations and topmost stack frames over a single lasso iteration implies periodicity.

To begin, we show that the existence of an ultimately periodic trace $\mu \cdot v^\omega$ implies the existence of an ultimately periodic trace $\mu' \cdot v'^\omega$ in which the sequence of global valuations (and topmost procedure-stack frames) of each thread at context-switch points repeat in each iteration of the lasso v' .

3.1 Annotated Traces

For a configuration $c = \langle g, \sigma \rangle$ of a program P , we write $c[g := g']$ to denote the configuration $c = \langle g', \sigma \rangle$. An *annotated trace* $\bar{\pi}$ of a program P is a sequence $\bar{\pi} = \langle g_i, \tau_i, \pi_i, g'_i, \tau'_i \rangle_{i=1..k}$ —where each $g_i, g'_i \in \text{Vals}$ are global valuations, $\tau_i, \tau'_i \in \text{Tids} \rightarrow (\text{Locs} \times \text{Vals})$ are thread-to-frame mappings, and π_i is a thread context—for which there exist local-state maps $\sigma_1, \sigma'_1, \dots, \sigma_k, \sigma'_k : \text{Tids} \rightarrow (\text{Locs} \times \text{Vals})^+$ of configurations $c_1, c'_1, \dots, c_k, c'_k$ where for each $1 \leq i \leq k$:

- $c_i = \langle g_i, \sigma_i \rangle$ and $c'_i = \langle g'_i, \sigma'_i \rangle$,
- $\sigma_i(t) = \tau_i(t) \cdot w_i$ and $\sigma'_i(t) = \tau'_i(t) \cdot w'_i$ for each thread $t \in \text{Tids}$, for some w_i, w'_i ,
- each c'_i is reachable from c_i via the trace π_i , and
- $c_{i+1} = c'_i[g := g_{i+1}]$ for $i < k$.

We say the annotated trace $\bar{\pi}$ is *valid* when $c_{i+1} = c'_i$ for $1 \leq i < k$. The definitions applying to traces are lifted naturally to annotated traces.

Lemma 1. *There exists an ultimately periodic trace $\mu \cdot v^\omega$ from a configuration c in a program P iff there exists a valid annotated ultimately periodic trace $\bar{\mu} \cdot \bar{v}^\omega$ from c in P .*

As we are mainly concerned with annotated traces, we usually drop the bar-notation, writing, e.g., π to denote an annotated trace $\bar{\pi}$, and use “trace” to mean “annotated trace.”

3.2 Compositional Detection of Periodic Traces

In the following we reduce the detection of valid ultimately periodic traces to the detection of ultimately periodic traces for each individual thread $t \in \text{Tids}$. Let $\pi = \mu \cdot v^\omega$ be a valid ultimately periodic trace which divides μ and v , resp., into $k_1 \in \mathbb{N}$ and $k_2 \in \mathbb{N}$ contexts, indexed by $I^\mu \subseteq \mathbb{N}$ and $I^v \subseteq \mathbb{N}$, as $\mu = \langle g_i, \tau_i, \mu_i, g'_i, \tau'_i \rangle_{i \in I^\mu}$ and $v = \langle g_i, \tau_i, v_i, g'_i, \tau'_i \rangle_{i \in I^v}$. We construct an ultimately periodic trace π_t in which only t is active. Roughly speaking, the constructed trace π_t corresponds to the projection of π on the set of t -labeled transitions. Given the global values g_i and g'_i seen at the beginning and end of each context i of thread t , the trace π_t can be computed in complete isolation: we simply resume the i th context of thread t with the global value g_i , and ensure g'_i is encountered at the end of the i th context. Supposing thread t executes in the contexts

indexed by $I_t^\mu \subseteq I^\mu$, along the stem μ , and in the contexts indexed by $I_t^\nu \subseteq I^\nu$ along the lasso ν , we define the *thread-periodic trace* for t as $\pi_t \stackrel{\text{def}}{=} \mu_t \cdot \nu_t^\omega$, where

$$\mu_t = \langle g_i, \tau_i(t), \mu_i, g'_i, \tau'_i(t) \rangle_{i \in I_t^\mu} \quad \nu_t = \langle g_i, \tau_i(t), \nu_i, g'_i, \tau'_i(t) \rangle_{i \in I_t^\nu}$$

For the thread-periodic trace π_t of t , we associate two sequences $SI(\pi_t) = \langle g_i, g'_i \rangle_{i \in I_t^\mu}$ and $LI(\pi_t) = \langle g_i, g'_i \rangle_{i \in I_t^\nu}$ of global valuation pairs encountered at the beginning and end of each context, called, resp., the *stem and lasso interfaces*; the sizes of interfaces are bounded by the number of contexts: $|SI(\pi_t)| \leq k_1$ and $|LI(\pi_t)| \leq k_2$.

We define the *shuffle* of a sequence set S inductively as $\text{shuffle}(\{\varepsilon\}) = \{\varepsilon\}$, and $\text{shuffle}(S) = \bigcup \{s_1 \cdot \text{shuffle}(S') : s_1 s_2 \dots s_j \in S \text{ and } S \setminus \{s_1 \dots s_j\} \cup \{s_2 \dots s_j\} = S'\}$; for instance, $\text{shuffle}(\{s_1 s_2, s_3\}) = \{s_1 s_2 s_3, s_1 s_3 s_2, s_3 s_1 s_2\}$. We say the thread interface sets S and L are *compatible* when there exists $s_1 \dots s_{k_1} \in \text{shuffle}(S)$ and $s_{k_1+1} \dots s_{k_1+k_2} \in \text{shuffle}(L)$ where each $s_i = \langle g_i, g'_i \rangle$, and $g'_i = g_{i+1}$ for $0 < i < k_1 + k_2$, and $g'_{k_1+k_2} = g_{k_1+1}$. Extending this definition, we say a set $\{\pi_t : t \in \text{Tids}\}$ of thread-periodic traces is *compatible* if and only if $\{SI(\pi_t) : t \in \text{Tids}\}$ and $\{LI(\pi_t) : t \in \text{Tids}\}$ are compatible.

Lemma 2. *If there exists a compatible set of thread-periodic traces $\{\pi_t : t \in \text{Tids}\}$ of a program P , then there exists a valid ultimately periodic trace $\pi = \mu \cdot \nu^\omega$ of P . Moreover, μ and ν are, resp., $\sum_{t \in \text{Tids}} |SI(\pi_t)|$ and $\sum_{t \in \text{Tids}} |LI(\pi_t)|$ context-bounded.*

Lemma 2 suggests a compositional algorithm to detect ultimately periodic valid traces. As each trace is constructed from a straight-forward composition of thread-periodic traces, we need simply to compute a compatible set of thread-periodic traces. We thus reduce the detection of valid ultimately periodic traces to computing (finite) compatible thread interface sets $\{S_t : t \in \text{Tids} \text{ and } |S_t| \leq k_1\}$ and $\{L_t : t \in \text{Tids} \text{ and } |L_t| \leq k_2\}$, and ensure the existence of, for each thread $t \in \text{Tids}$, a thread-periodic trace π_t such that $SI(\pi_t) = S_t$ and $LI(\pi_t) = L_t$.

3.3 From Thread-Periodic Traces to Sequential Reachability

Section 3.2 reduced the problem of finding periodic executions to that of computing thread interfaces. Now we demonstrate that thread interfaces can be computed by state-reachability in sequential programs. For the remainder of this section we fix an initial configuration c_0 of a program P , and a thread-periodic trace $\pi_t = \mu_t \cdot \nu_t^\omega$ of a thread t .

We know that the thread period trace π_t repeats the same sequence of actions per period over and over indefinitely. It follows that although during each period the size of t 's frame stack may increase and decrease due to procedure calling and returning, the *net size* of t 's frame stack must not be decreasing—otherwise t cannot repeat ν_t indefinitely. This implies that there exists a sequence $f_1 f_2 \dots$ of t 's procedure frames—each $f_i \in (\text{Locs} \times \text{Vals})$ encountered in the i th period—which are never returned from; we call these frames the *keyframes* of t . Since we repeat the same sequence of calls and returns along each period, we can assume w.l.o.g. that each keyframe f_i is the procedure frame encountered at the beginning of the same context shift in ν_t with $0 \leq \text{shift} < |LI(\pi_t)|$. Furthermore, we know that these keyframes correspond to the same procedure frame f (from definition of value annotated traces).

In order to check that f is a keyframe (i.e., never removed from the stack), we check that from a configuration where the stack contains only the frame f , we can reach a configuration with topmost frame f after executing the trace v_t (modulo rotation). We know also that executing the trace μ_t followed by the first shift contexts in v_t will result in a configuration with topmost frame f . This is exactly what is defined below:

Feasibility: Let $SI(\pi_t) = \langle g_i, g'_i \rangle_{i \in I_t^\mu}$ and $LI(\pi_t) = \langle g_i, g'_i \rangle_{i \in I_t^\nu}$ be the given stem and lasso interfaces. We say that $\langle SI(\pi_t), LI(\pi_t) \rangle$ is *feasible* if there are a frame f , a natural number `shift` with $0 \leq \text{shift} < k_1$, and a sequence of configurations $c_1, c'_1, \dots, c_m, c'_m$ with $m = (k_1 + k_2 + \text{shift})$ such that, for every $1 \leq j \leq m$:

- c'_j is reachable from c_j via a trace of the thread t .
- The global valuation in c_j and c'_j are g_i and g'_i with $i = j(\text{mod } k_1 + k_2) + k_2 + 1$.
- The stack in c'_{j-1} and c_j are the same when $j \neq (k_2 + \text{shift} + 1)$ with $c'_0 = c_0$.
- The stack in $c'_{k_2 + \text{shift} + 1}$ contains only the frame f . Moreover, the topmost frame in $c'_{k_2 + \text{shift}}$ and c'_m is precisely f .

Lemma 3. *If the thread trace π_t is periodic, then $\langle SI(\pi_t), LI(\pi_t) \rangle$ is feasible.*

Now, we can show if there is a thread trace π'_t of t from a configuration containing the keyframe f , satisfying the interface L_t , and reaching a configuration whose topmost frame is precisely f , then this thread trace can be executed infinity often. This means that π'_t can be considered as a lasso trace of t whose lasso interface is precisely L_t . On the other hand, if there is a thread trace π''_t of t from the initial configuration to a configuration whose topmost keyframe is precisely f while respecting the stem interface S'_t (which is the concatenation of S_t and the first (`shift`)-elements of L_t) then $\pi''_t \cdot \pi'_t$ can be considered as a stem trace for the lasso trace π'_t .

Lemma 4. *Given a compatible interface sets $\{S_t : t \in \text{Tids}\}$ and $\{L_t : t \in \text{Tids}\}$ such that $\langle S_t, L_t \rangle$ is feasible for each $t \in \text{Tids}$, we can construct compatible thread-periodic traces $\{\pi_t : t \in \text{Tids}\}$ such that $|SI(\pi_t)| = |S_t| + |L_t|$ and $|LI(\pi_t)| = |L_t|$ for each $t \in \text{Tids}$.*

The lemmata above suggest the following procedure: first, guess compatible interfaces $\{S_t, L_t : t \in \text{Tids}\}$, then check feasibility of each $\langle S_t, L_t \rangle$. Observe that checking the feasibility of each given pair $\langle S_t, L_t \rangle$ boils down to solving reachability problems in the sequential program describing the behavior of the thread t . Section 4 concretizes this algorithm in a code-to-code reduction to sequential program analysis.

3.4 Encoding Fairness

By our definitions in Section 2 any blocked thread must be waiting to acquire a held lock. This leads to the following characterization of strongly fair ultimately periodic traces: for each thread $t \in \text{Tids}$, either

Case 1. The lasso contains at least one transition of t , or

Case 2. The thread t is blocked throughout the lasso, waiting to acquire some lock $x \in \text{Locks}$; this further implies that

Cond. 1 the lock x may not be released during the lasso by any thread,

Cond. 2 the lock x must be held by another thread at the beginning of the lasso,

and

Cond. 3 t remains at the control location of the acquire of x throughout the lasso.

These conditions characterize strongly fair ultimately periodic computations. We ensure these conditions are met by extending the notion of interfaces to include the set $X \subseteq \text{Locks}$ of locks which are held throughout the lasso. Then, we must ensure locally per thread that any $x \in X$ is not released during the lasso (Cond. 1), and that some thread holds x when entering the lasso (Cond. 2); additionally, we allow any thread attempting to acquire some $x \in X$ to execute no further action. (Observe that if the lock x is released during the lasso by any thread (see Cond. 1 of Case 2) then the resulting ultimately periodic computation is not strongly-fair since the thread t is infinitely often enabled and does not infinity often fire a transition.) Weak fairness can be similarly characterized using a set $Y \subseteq \text{Locks}$ of locks which are held at some point during the lasso; we then ensure that each $y \in Y$ is either held at the beginning of the lasso, or acquired at some point during the lasso.

4 Reduction to Sequential Program Analysis

The compositional analysis outlined in Section 3 reduces (context-bounded) fair periodic non-termination to state-reachability in sequential programs. Given thread stem and lasso interfaces, and the set of locks held throughout the lasso, the feasibility of each interface is computed separately, per thread. In this section we describe how to implement this reduction by a code-to-code translation to sequential programs with an assertion which fails exactly when the source program has a strongly fair ultimately periodic execution whose stem and lasso satisfy a given context bound.² Figure 2 lists our translation in full.

Essentially, we introduce a `Main` procedure for the target program which executes each thread one-by-one using an initially-guessed sequence of global valuations stored in `Stem0` and `Lasso0`. For each thread t , we guess the number—stored in `shift`—of contexts following the k_1 st context until t 's keyframe is encountered on Line 18, and begin executing t 's main procedure `Main[t]` on Line 21. Initially, the values stored in `Stem` and `Lasso` are the values seen at the beginning of each context of the first thread during, resp., the stem and repeating lasso. After execution of the i th thread, the values of `Stem` and `Lasso` are the values seen at the end of each context of the i th thread, and at the beginning of each context of the $(i+1)$ st thread. Accordingly, after the execution of the final thread, the values seen at the end of each context must match the values guessed at the beginning of the following contexts of the first thread, according to the round-robin order; the assumptions on Lines 22–27 ensure these values match.

The execution of each thread thus acts simply to compute its interface. As the keyframes of different threads may be encountered at different points along the lasso,

² Technically we consider bounded round-robin thread schedules rather than bounded context switch. Though in principle the two notions are equivalent for a fixed number of threads—i.e., any k -context execution takes place within k rounds, and any k -round n -thread execution takes place in kn contexts [15]—ensuring interface compatibility is simpler assuming round-robin.

the length of each thread’s stem varies. Our translation computes for each thread a stem long enough (at most $k_1 + k_2 - 1$ contexts) to cover the stem of any thread. Since each thread’s repeating sequence may begin as soon as the k_1 st context, the stem and lasso computation may overlap. Our translation maintains the invariant that the `Stem` (resp., `Lasso`) values are active exactly when $k_1 \neq \perp$ (resp., $k_2 \neq \perp$). Reads and writes to shared variables (on Lines 47–57) read and write to both `Stem` and `Lasso` as they are active.

Our translation also adds code at every potential context switch point (Lines 71–96). Initially, the context counters k_1 and k_2 are incremented nondeterministically and synchronously (the block starting at Line 74). Then, at Line 78 we check whether the thread’s keyframe has been encountered for the first time, and if so make a snapshot of the local valuation and program location, and activate the lasso; later along, at Line 90 we validate the snapshot when returning to the same keyframe (perhaps with a larger procedure stack). At some point in between, at Line 86, the stem becomes inactive. We ensure using the local variable `bottom` that the keyframe in which a thread begins repeating is never returned from.

We ensure strong fairness using an auxiliary vector of Boolean constants `waited`, one per lock $x \in \text{Locks}$, indicating the set of locks which are held throughout the lasso. According to Section 3.4, we ensure each waited lock is held at the beginning of the lasso (Lines 28–30) and not released during the lasso (Line 68), and allow attempted acquires to abort (Line 63).

Lemma 5. *The program $((P))^{k_1, k_2}$ violates its assertion if P has a strongly-fair ultimately periodic round-robin execution with $k_1 \in \mathbb{N}$ and $k_2 \in \mathbb{N}$, resp., stem and lasso rounds; if $((P))^{k_1, k_2}$ violates its assertion then P has a strongly-fair ultimately periodic round-robin execution with $k_1 + k_2$ and k_2 , resp., stem and lasso rounds.*

5 Experimental Evaluation

We have implemented our analysis, based on the code-to-code translation presented in Section 4. Our prototype tool, called `MUTANT`³, takes as input a program written in the BOOGIE intermediate verification language [2]. Though normally a rich sequential language with recursive procedures, integers, maps, and algebraic datatypes, we have extended BOOGIE with thread-creation and atomic blocks, which we use to model shared-memory multithreaded programs with synchronization operations. Given a bound $K \in \mathbb{N}$ (where $K = k_1 + k_2$), `MUTANT` outputs an assertion-annotated sequential BOOGIE program. We feed the resulting program to our SMT-based bounded model checker `CORRAL` [14]. `MUTANT` has support for strong fairness, and does not falsely detect nonterminating executions in the program of Figure 1a, for instance.

As an initial example to demonstrate `MUTANT`’s effectiveness, we consider a *try-lock* based algorithm for the *dining philosophers* problem. This program involves N locks and N threads, each of which executes the code shown in Figure 3a. Each philosopher tries to acquire two locks. TryLock is a non-blocking synchronization operation that returns `true` when the lock is successfully acquired, otherwise it returns `false`. If

³ `MUTANT` stands for MultiThreAded Non Termination.

```

// translation of
// var g: T
var Stem[k1+k2-1]: T
var Lasso[k2]: T
5 var Local: T
var Location: Locs
var shift: N<k2 ∪ {⊥}
const waited[Locks]: B
var k1: N ∪ {⊥}
10 var k2: N ∪ {⊥}

proc Main()
    const Stem0 := ∗;
    const Lasso0 := ∗;
15    Stem := Stem0;
    Lasso := Lasso0;
    foreach t in Tids do
        shift := ∗;
        k1 := 0;
20        k2 := ⊥;
        call Main[t] ();
    assume
        Stem[0..k1+k2-3]
        = Stem0[1..k1+k2-2];
25    assume
        Lasso[0..k2-2]
        = Lasso0[1..k2-1];
    assume ∀x ∈ Locks.
        x(Lasso[0])
30        ⇔ waited[x];
    assert false;
    return

```

```

// translation of
// proc p (var l: T) s
35 proc p (var l: T,
            bottom: B) s
    // translation of
    // call x := p e
40 call x := p (e, ∗)
    // translation of
    // return e
45 return e

```

```

// translation of
// variable read x := g
assume Stem[k1]
50      = Lasso[k2];
x := Stem[k1]
x := Lasso[k2];

```

```

// translation of
// variable write g := e
55 Stem[k1] := e;
Lasso[k2] := e

```

```

// translation of
// acquire x
60 if shift = ⊥ ∧ ∗
        ∧ waited[x] then
            abort;
        acquire x

```

```

65 // translation of
// release x
assume k2 ⇒ !waited[x];
release x
70 // translation of
// (implicit) yield
// at location 'loc'
while ∗ do
75    k1 := k1 + 1;
    k2 := (k2+1) mod k2;
    if k1 = k1+shift
        ∧ k2 = ⊥ then
80        // begin the lasso
        assume bottom;
        k2 := shift;
        Local := 1;
        Location := loc;
85    if k1 ≥ k1+k2-1 then
        // end the stem
        k1 := ⊥;
90    if k2 = shift
        ∧ k1 = ⊥ then
        // end the lasso
        assume Local = 1;
        assume Location = loc;
95    // exit to main
    abort;

```

Fig. 2. The sequential translation $((P))^{k_1, k_2}$ of a multithreaded program P . We assume that statements which evaluate undefined expressions (i.e., using \perp in arithmetic or array indexing) are simply skipped, and that no statement both reads and writes to g . The expression \star nondeterministically evaluates to any well-typed value, and the **assume** e statement proceeds only when e evaluates to **true**. The **abort** statement discards the procedure stack and returns control to Main .

a philosopher acquires the left lock but is not able to acquire the right lock, then he releases the left lock and tries again. A philosopher terminates when he is able to acquire both locks (Line 10). This program has a fair non-terminating execution for each $N \geq 2$, namely where each philosopher first acquires their left lock, then upon seeing their right lock unavailable, they release their left lock. MUTANT is able to automatically detect this execution for each value of N with $K = 2$; we report running times in Figure 3d. Note that while this execution requires all N threads to participate, each thread only uses a fixed number of context switches in each period of the lasso. Though the state-space of the program grows exponentially with N , Figure 3d demonstrates that MUTANT scales sub-exponentially. Though the program has unfair non-terminating executions—e.g., where one philosopher acquires a lock and ceases to participate further, while the others continuously spin waiting to acquire both their locks—MUTANT correctly does not report any such unfair non-terminating executions.

<pre> 1 // An array of N locks 2 var Lock[N]: mutex 3 4 proc Philosopher(n: int) 5 var left := Lock[n]; 6 var right := Lock[(n+1)%N]; 7 while true do 8 if TryLock(left) 9 if TryLock(right) 10 break 11 else 12 ReleaseLock(left); 13 ReleaseLock(right); 14 ReleaseLock(left); 15 return </pre>	<pre> 1 proc Thread1() 2 var v1 := *; 3 add(v1); 4 flag := false; 5 return </pre>	<pre> 1 while e1 do 2 timeout := false; 3 if * and e2 then 4 timeout := true; 5 break </pre>
(a)	(b)	(c)

<pre> 7 proc Thread2() 8 while flag do 9 var v2 := *; 10 if * then 11 add(v2) 12 else 13 remove(v2); 14 return </pre>		<pre> N=2 3 4 5 6 2.1s 3.43s 6.13s 8.94s 21.91s N=7 8 9 10 15.79s 30.77s 31.66s 43.54s </pre>
(a)	(b)	(d)

Fig. 3. (a) TryLock based dining philosophers. (b) A concurrent client operating on an OptimisticList. (c) Modeling timeout. (d) Running time of MUTANT on the dining philosophers example. As our verifier is based on the Z3 SMT solver, running times may increase non-uniformly with N due to Z3’s internal heuristics, which may vary widely across different instances.

As a second example we consider the concurrent OptimisticList algorithm from Section 9.6 of Herlihy and Shavit [9], supporting concurrent insertions and deletions on sorted lists using optimistic concurrency control. Our BOOGIE encoding spans roughly 250 lines. In order to determine whether each operation is guaranteed to terminate in the presence of an environment performing arbitrary list operations, we wrote the two-thread driver of Figure 3b. While the first thread tries to insert an element, the second thread continuously fires add and remove operations with arbitrary arguments. The shared variable `flag` ensures that the second thread terminates when the first thread does. Though not shown, the driver also initializes the list with a few arbitrary elements.

This program has the following fair non-terminating execution, similar in spirit to that in Figure 3c: first, the `add` operation of `Thread1` selects a position in the (sorted) list where to insert a value v_1 , say between consecutive nodes with values a and b (i.e., such that $a < v_1 < b$). Then the second thread picks a value v_2 , such that $a < v_2 < b$, and inserts. When the first thread then sees that list has been modified at the position it was about to insert, it retries the `add` operation. Meanwhile, the second thread fires a `remove` operation and deletes v_2 . This program then reencounters the initial configuration, and the `add` operation has not succeeded. MUTANT finds this execution with three contexts per thread in 44 seconds. Interesting to note is that even though this program may use infinite-domain data values, there remains nevertheless an execution that loops back exactly to the configuration. One slightly tricky aspect of this example is modeling memory allocation: because the second thread allocates and removes a list node in each period, we must explicitly free the removed node in order to reencounter the same configuration at the end of the lasso. As future work, using a more abstract notion of heap equality could simplify this aspect.

As a third example we consider a algorithm developed by our colleagues [20] that enables programmers to write assertions which are checked *continuously and concurrently* with the actual program, in similar spirit to *asynchronous assertions* [1]. One salient

feature of this algorithm is that it is non-blocking, i.e., the evaluation of the asserted expressions does not block other threads from making progress. We coded the algorithm, and two variations with possible non-termination bugs, in roughly 230 lines of BOOGIE code. In each of the potentially-buggy variations we found a non-terminating execution where incorrect assertion evaluations led to livelock. To our surprise, we also found a non-terminating execution in our supposedly-correct variation. After consulting with the developers, the problem turned out to be in our modeling. To understand the problem, consider the code in Fig. 3c. MUTANT detected non-termination by skipping the `then`-branch in each iteration of the lasso. (The actual non-termination found by MUTANT required concurrent reasoning, even though the lasso only involved one thread.) However, the intention of the designers was that this branch represents an actual time out reflecting a timer running down to zero. We corrected this modeling by ensuring that the above choice must evaluate to `true` at least once within the lasso. This is similar to enforcing Condition 1, Case 2 of strong fairness in Section 3. Nonetheless, MUTANT’s output is still valuable: it says that if the time out is not implemented correctly, then the program may enter a livelock.

MUTANT is able to determine the absence of periodic nontermination bugs in the corrected variation with up to 3 contexts per thread in 402 seconds. MUTANT also detects nonterminating executions in the three buggy variations in 11, 21, and 36 seconds. These experiments demonstrate that MUTANT is effective on real-world algorithms.

6 Related Work

Our work follows the line of research on compositional reductions from concurrent to sequential programs. The initial so-called “sequentialization” [22] explored multithreaded programs up to one context-switch between threads. Following Qadeer and Rehof [21]’s generalization of context-bounding to an arbitrary number of context switches, Lal and Reps [15] later proposed a sequentialization to handle a parameterized amount of context-switches between a statically-determined set of threads executing in round-robin order. La Torre et al. [12] extended the approach to handle programs parameterized by an unbounded number of statically-determined threads, and shortly after, Emmi et al. [6] further extended these results to handle an unbounded amount of dynamically-created tasks. Bouajjani et al. [3] pushed these results even further to a sequentialization which attempts to explore as many behaviors as possible within a given analysis budget. The compositional analyses resulting from each of these sequentializations however only consider finite executions, and are thus incapable of establishing liveness properties.

Although much previous work has been done for proving termination and detecting non-termination in sequential programs—for instance, Cook et al. [4] discover ranking functions to prove termination of sequential programs, and Gupta et al. [8] use concolic execution to detect non-terminating executions in sequential programs—relatively little attention has been paid to multithreaded programs, where interesting non-terminating executions often have little to do with possible divergence of data values. Though Cook et al. [5] have extended TERMINATOR to multithreaded programs, their analysis is oriented to proving termination; failure to prove termination does not generally indicate the

existence of a non-terminating execution. More recently Popeea and Rybalchenko [19] have developed compositional techniques to prove termination in multithreaded programs, though again, their approach does not certify the existence of non-terminating executions. Because both of these techniques focus on establishing a proof of termination, they necessarily consider over-approximations of concurrent programs, whereas our technique looks at an under-approximation to find counterexamples faster.

Musuvathi and Qadeer [18] consider liveness properties in multithreaded programs, but their approach is based on systematic testing, and thus behavioral coverage is limited by test harnesses and concrete input values. Moreover, their approach is stateless (i.e., they never store states during the execution of the program), hence they can only detect possible non-termination by identifying lengthy executions.

In the most closely related work of which we are aware, Morse et al. [16] propose a compositional LTL model checking technique for multithreaded programs based on context-bounding. As far as we can tell, their technique (a) does not ensure non-terminating executions are fair, (b) does not consider lassos in which multiple recursive threads interfere, and (c) requires very high context-bounds to capture synchronized interaction between the program and a monitor Büchi automaton.

7 Conclusion

We have developed a compositional algorithm for detecting fair ultimately periodic executions in recursive multithreaded programs by bounding the number of context-switches in each repeating period. Our approach reveals a simple-to-implement code-to-code translation, which reduces the problem to finding assertion violations in recursive sequential programs; consequently we leverage existing sequential analysis algorithms.

Our approach can be used to encode other linear temporal logic conditions besides non-termination, e.g., response properties. Though for specific classes of formulae/properties efficient encodings are possible, a sequentialization parameterized by arbitrary linear temporal logic formulae must essentially construct the product of the input program with an arbitrary Büchi automaton; the encoding of this (synchronous) product as a sequential program may not be as succinct.

In this work, discovering ultimately periodic executions is done by detecting repeated state valuations. This notion of repeatability is complete for programs manipulating finite data, but is not complete in general. Still, this notion is actually relevant in many practical cases, since non-termination bugs in concurrent programs are often due to non-state-changing retry mechanisms. In the case of infinite data domains periodic executions may exhibit, for instance, ever increasing counter values; there a notion of repeatability more relaxed than state-equality may be necessary. This notion however, contrary to the one we consider here, would have to account for the actions encountered during the lasso. Ensuring repeatability may be complex to define and check, depending on the data domains and the nature of program operations.

References

- [1] Aftandilian, E., Guyer, S.Z., Vechev, M.T., Yahav, E.: Asynchronous assertions. In: OOPSLA 2011: Proc. 26th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 275–288. ACM (2011)

- [2] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
- [3] Bouajjani, A., Emmi, M., Parlato, G.: On Sequentializing Concurrent Programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011)
- [4] Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI 2006: Proc. ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation, pp. 415–426. ACM (2006)
- [5] Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: PLDI 2007: Proc. ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation, pp. 320–330. ACM (2007)
- [6] Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: POPL 2011: Proc. 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 411–422. ACM (2011)
- [7] Garg, P., Madhusudan, P.: Compositionality Entails Sequentializability. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 26–40. Springer, Heidelberg (2011)
- [8] Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL 2008: Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 147–158. ACM (2008)
- [9] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
- [10] Kozen, D.: Lower bounds for natural proof systems. In: FOCS 1977: Proc. 18th Annual Symp. on Foundations of Computer Science, pp. 254–266. IEEE Computer Society (1977)
- [11] La Torre, S., Madhusudan, P., Parlato, G.: Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
- [12] La Torre, S., Madhusudan, P., Parlato, G.: Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)
- [13] Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
- [14] Lal, A., Qadeer, S., Lahiri, S.: Corral: A Solver for Reachability Modulo Theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
- [15] Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009)
- [16] Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Context-Bounded Model Checking of LTL Properties for ANSI-C Software. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 302–317. Springer, Heidelberg (2011)
- [17] Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multi-threaded programs. In: PLDI 2007: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 446–455. ACM (2007)
- [18] Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: PLDI 2008: Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation, pp. 362–371. ACM (2008)
- [19] Popeea, C., Rybalchenko, A.: Compositional Termination Proofs for Multi-threaded Programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 237–251. Springer, Heidelberg (2012)

- [20] Qadeer, S., Musuvathi, M., Burnim, J.: Personal communication (January 2012)
- [21] Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
- [22] Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI 2004: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 14–24. ACM (2004)
- [23] Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
- [24] Yen, H.C.: Communicating processes, scheduling, and the complexity of nondeterminism. Mathematical Systems Theory 23(1), 33–59 (1990)

Lock Removal for Concurrent Trace Programs *

Vineet Kahlon¹ and Chao Wang²

¹ NEC Laboratories America, Princeton, NJ 08540

² Department of ECE, Virginia Tech, Blacksburg, VA 24061

Abstract. We propose a trace-based concurrent program analysis to *soundly* remove redundant synchronizations such as locks while preserving the behaviors of the concurrent computation. Our new method is computationally efficient in that it involves only *thread-local* computation and therefore avoids interleaving explosion, which is known as the main hurdle for scalable concurrency analysis. Our method builds on the partial-order theory and a unified analysis framework; therefore, it is more generally applicable than existing methods based on simple syntactic rules and *ad hoc* heuristics. We have implemented and evaluated the proposed method in the context of runtime verification of multithreaded Java and C programs. Our experimental results show that lock removal can significantly speed up symbolic predictive analysis for detecting concurrency bugs. Besides runtime verification, our new method will also be useful in applications such as debugging, performance optimization, program understanding, and maintenance.

1 Introduction

Concurrent programs are notoriously difficult to analyze due to their behavioral complexity resulting from the often extremely large number of thread interleavings. This renders comprehending all the possible ways in which threads interact a difficult problem. As a result, programmers often take a defensive stance and label large sections of code as critical sections. This may result in the addition of redundant locks, both degrading performance and making program modeling, analysis, and understanding difficult. The situation is particularly severe in trace-based concurrent program analysis. When focusing on a concrete execution trace rather than the entire program, we often find significantly more redundant locks, i.e. locks that are not completely redundant in the whole program may become redundant when the analysis is restricted to a trace.

Although there exist some methods for identifying redundant synchronizations in Java and C programs [34622130], e.g. as part of the compiler’s performance optimization, they are all based on very simple syntactic rules and *ad hoc* heuristics. Since these methods are based on matching patterns rather than analyzing the program semantics, they do not lead to a generally applicable framework. Indeed, most of them handle only the simple case of *effectively thread-local* objects, i.e. locks that are declared as globally visible but are accessed only by one thread throughout the execution. For the many truly shared but still redundant locks, these existing methods are not effective.

We address this limitation by introducing a new and more generally applicable lock removal algorithm. Our method is generally applicable since it can remove not only the

* Chao Wang was supported in part by the NSF CAREER award CCF-1149454.

effectively *thread-local* locks but also the *truly shared* redundant locks. Our method is also efficient since it is based on a compositional analysis that involves only thread-local computation. Our method is sound in that it can guarantee preservation of the behavior of the original computation.

In formulating our lock removal strategy, we start from the classical notion of a concurrent computation as a happens-before relation on the shared variable accesses or, equivalently, as a set of partial orders. Two interleavings are equivalent if they induce the same partial order of shared variable accesses. Since removing locks lifts the corresponding mutual exclusion constraints, some previously infeasible thread interleavings may become feasible. Thus there is a danger for lock removal to introduce new program behaviors. To address this problem, we make sure that new interleavings are added by lock removal only if they do not add new partial orders. This leads to the formulation of the *behavior preservation* theorem, which is a main contribution of this paper.

Another main contribution is the set of *efficiently checkable* conditions under which the behavior preservation is guaranteed. They reduce the semantic check of behavior preservation to a simple static check of the feasibility of transitions between global control states. This is significant because it allows us to avoid enumerating the often astronomically large number of thread interleavings. Our method is thread-modular in that it does not require inspecting the interleaved parallel composition of threads. In addition, our focus on a concrete execution trace is also crucial in keeping the method scalable. The concrete execution trace provides the exact memory addresses that are accessed by each thread, thereby giving us the precise points-to information of lock pointers, together with information about the actual array fields accessed, etc.

Trace-based concurrent program analysis has obvious applications not only in runtime verification, but also in debugging, just-in-time (JIT) optimization, program understanding, and maintenance. An important feature of trace-based analysis is that the trace program has finitely many threads and a fixed set of named locks. Although the whole program may have pointers, loops, recursion, and dynamic thread creation, in the trace program, each thread is reduced to a bounded straight-line path. Most of the complications common to static program analysis are avoided because, during the concrete execution, branching decisions at if-else statements have been made, function calls have been inlined, loops have been unrolled, and recursions have been applied. The only remaining source of nondeterminism comes from thread interleaving.

We have implemented the proposed method in a runtime verification platform called *Fusion*, where the underlying bug detection algorithm uses an SMT-based symbolic analysis. Since redundant locks can introduce a large set of synchronization constraints during the modeling and checking phases, their presence often significantly increases the cost of the symbolic analysis. Our lock removal method has been used to remove these redundant locks. Our experiments on a set of public Java and C programs showed a significant reduction in the number of locks, which in turn led to a significant speedup in the subsequent symbolic analysis.

To sum up, this paper has made the following two contributions: (1) formulating the general framework of behavioral preservation to soundly remove redundant locks; and (2) proposing a set of efficiently checkable conditions based on the thread-local computation of lock access patterns.

The remainder of this paper is organized as follows. In Section 2, we use two examples to illustrate both the benefit and challenges of lock removal. In Section 3, we illustrate our main ideas. In Section 4, we present a set of efficiently checkable conditions. In Section 5, we demonstrate the application of our algorithm on the running example. Our experimental results are presented in Section 6. We review the related work in Section 7 and give our conclusions in Section 8.

2 Motivation

The main driving application in this paper is runtime predictive analysis [12][25][11][23][29][19], which is a promising method for detecting concurrency bugs by analyzing an execution trace. In other words, even if the given test execution is not erroneous, but if an alternative interleaving of the events of that trace can trigger a failure, runtime predictive analysis will be able to detect it. Since a concurrent program often has a very large number of sequential paths and thread interleavings, statically analyzing the whole program is often extremely difficult. In such cases, runtime predictive analysis offers a good compromise between runtime monitoring and full-fledged model checking.

Runtime predictive analysis typically has three steps: (1) run a test of the concurrent program to obtain an execution trace; (2) run a sound static analysis of the trace to compute all the *potential* violations, e.g. deadlocks and race conditions; (3) for each potential violation, build a precise predictive model to decide whether the violation is feasible. The main scalability bottleneck is step 3 wherein the feasibility check needs to explore all possible interleavings of the trace events. Although the problem in step 3 can be solved by an efficient symbolic analysis [29][19], redundant locks in the trace program can unnecessarily increase the cost of this analysis, since they can lead to a large number of locking constraints that need to be modeled and checked. Our lock removal method can cut down on the number of unnecessary locking constraints, therefore resulting in significant performance improvement in the subsequent analysis.

$T_1 0 \{$ 0a: —; 1a: lock(A); 2a: a[1]++; 3a: unlock(A); 4a: a[2]++; 5a: lock(A); 6a: lock(B); 7a: a[3]++; 8a: unlock(B); 9a: unlock(A); 10a: sh++; $\}$	$T_2 0 \{$ 0b: —; 1b: a[10]++; 2b: a[11]++; 3b: lock(A); 4b: lock(B); 5b: a[12]++; 6b: unlock(B); 7b: unlock(A); 8b: sh++; $\}$	$T'_1 0 \{$ 0a: —; 1a: lock(A); 2a: —; 3a: unlock(A); 4a: —; 5a: lock(A); 6a: lock(B); 7a: —; 8a: unlock(B); 9a: unlock(A); 10a: sh++; $\}$	$T'_2 0 \{$ 0b: —; 1b: —; 2b: —; 3b: lock(A); 4b: lock(B); 5b: —; 6b: unlock(B); 7b: unlock(A); 8b: sh++; $\}$	$T''_1 0 \{$ 0a: —; 1a: —; 2a: —; 3a: —; 4a: —; 5a: —; 6a: —; 7a: —; 8a: —; 9a: —; 10a: sh++; $\}$	$T''_2 0 \{$ 0b: —; 1b: —; 2b: —; 3b: —; 4b: —; 5b: —; 6b: —; 7b: —; 8b: —; $\}$
(a) original	(b) intermediate	(c) final			

Fig. 1. Example: removing redundant lock statements from a concurrent trace program

Consider the concurrent trace program in Fig. 1(a), which has two straight-line paths in threads T_1 and T_2 , respectively. The global variables are sh and array a . Suppose that the goal is to check whether locations $10a$ and $8b$ are simultaneously reachable (e.g. a data race), we need to decide whether there exists a valid interleaving of these trace statements along which T_1 and T_2 can reach $10a$ and $8b$, respectively.

First, note that precise knowledge of the memory accesses is available since the trace program is derived from a concrete execution. The knowledge can be used to cut down the number of shared accesses that need to be interleaved. For example, although $a[i]$ is a global variable, the entries of a accessed by the two threads in this particular trace program are all disjoint and can be treated as thread-local. In other words, we can use the runtime information to *slice away* the redundant statements. This can reduce the trace program in Fig. 1(a) to the one in Fig. 1(b).

Next, consider the program in Fig. 1(b). Since locks A and B now protect only thread-local statements, some of these lock statements may be redundant. We shall show in later sections that, for this particular example, these lock statements are all redundant and therefore can be removed while preserving the original program behavior. This reduction yields the simple trace program shown in Fig. 1(c) with only the shared variable accesses. Consequently, it becomes easy to decide the simultaneous reachability of $10a$ and $8b$.

Challenges in Lock Removal. The example in Fig. 1 may give a false impression that *locks protecting only thread-local operations can always be removed*. This is not true, as demonstrated by Fig. 2. In this example, variable $sh=0$ initially. The assertion at b_7 holds because, to get value 2, one has to execute $b_1 \dots b_3 \rightarrow a_1 \dots a_6 \rightarrow b_4 \dots b_7$, which is impossible since lock A is held by thread T_2 at b_3 , which prevents thread T_1 from acquiring the same lock at location a_2 . However, if we remove the lock/unlock statements at a_2 and a_4 – since they protect only thread-local operations – the assertion at b_7 may fail because the aforementioned interleaving is now allowed. This example highlights the fact that locks may play a key role in defining the set of allowed program behaviors even if they do not guard any global operation. It also shows that, without a rigorous concurrency analysis, *ad hoc* heuristics are often susceptible to subtle errors. We address this problem by proposing a generally applicable lock removal framework.

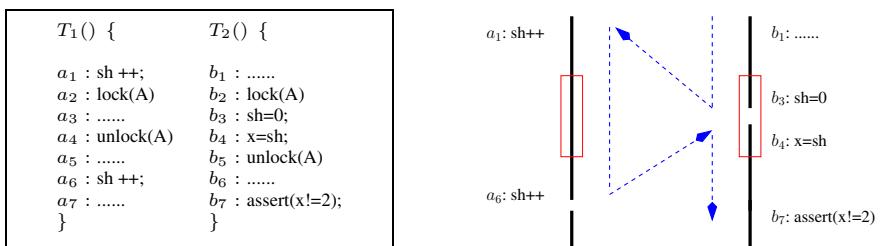


Fig. 2. Example: Assuming that $sh=0$ initially. The lock statements at a_2 and a_4 cannot be removed despite that they do not protect any shared access. Otherwise, assertion at b_7 may fail.

3 Lock Removal: The Core Idea

We say that a program \mathcal{P}' results from another program \mathcal{P} via lock removal if \mathcal{P}' is obtained from \mathcal{P} by converting some of the lock statements to `nop`. A lock statement in \mathcal{P} is considered as redundant if removing that statement does not alter the program behavior. Here the program behavior is defined as the set of interleaved computations that are allowed by the program semantics. Since lock statements impose mutual exclusion constraints, they restrict the thread interactions. By removing lock statements from \mathcal{P} , in general, we may allow the new program \mathcal{P}' to have more interleavings; on the other hand, it is impossible to remove any previously allowed interleavings in \mathcal{P} . Therefore, to preserve the program behavior, we only need to ensure that every newly added interleaving (allowed in \mathcal{P}' but not in \mathcal{P}) is equivalent, in some sense, to an existing interleaving in \mathcal{P} . In other words, lock removal is sound as long as it does not add new equivalence classes (of interleavings).

3.1 The Lock Removal Strategy

Since characterizing interleavings directly is cumbersome and computationally expensive, we rely on the standard notion of concurrent computations as happens-before relations on the shared variable accesses [2014]. That is, executing two operations from different threads that update the same memory location in different orders may lead to different results. Therefore, instead of preserving interleavings of all the statements, we focus on preserving the partial orders of shared variable accesses (reads and writes).

For a program \mathcal{P} comprised of the n threads T_1, \dots, T_n , a global control state s is a tuple (c_1, \dots, c_n) where c_i is a control location of T_i for all $i \in [1..n]$. In contrast to a *concrete* program state, denoted $s \in \mathbb{S}$, the global control state s is more *abstract* in that it tracks only the program counters but not the values of the program variables. Therefore s can be viewed as a set of concrete states. Since thread-local operations are invisible to the other threads, in the sequel we shall assume without loss of generality that the locations in (c_1, \dots, c_n) are all starting points of **global operations**, i.e. either shared reads/writes or lock acquisitions. This restriction can drastically cut down the number of global control states that need to be considered during our analysis. Note that if a thread is at location c_i , it means that the operation at c_i has not been executed yet.

Definition 1 (Visible Successor). For global control states s, s' in program \mathcal{P} , we say that s' is a visible successor of s iff there exist states $s \in \mathbb{S}$ and $s' \in \mathbb{S}'$ such that

- s' is reachable from s via a valid concurrent computation, and
- along this computation, the first operation is the only global operation.

Our lock removal strategy can be phrased as follows: Removing all lock statements such that no new visible successor is introduced to any global control state that is reachable from the initial state in \mathcal{P} . In other words, for each s , if we can preserve the set of global control states that s can transit to, the program behavior will be preserved.

Consider Fig. 2 as an example. For all transitions between two global control locations, e.g. from (a_2, b_3) to (a_6, b_3) , our lock removal strategy says that, if the transition

is not allowed by \mathcal{P} before lock removal, it should not be allowed by \mathcal{P}' either. Based on this strategy, the lock statements at a_2 and a_4 will be preserved, because removing them would make the infeasible transition in \mathcal{P} from (a_2, b_3) to (a_6, b_3) feasible in \mathcal{P}' .

3.2 Conservative Static Check

Although the lock removal strategy proposed so far is sound as well as generally applicable, computing the visible successors of a global control state is a challenging task, because the conditions in Definition II are semantic conditions. Checking the reachability between two concrete states s and s' would be too expensive in practice. To avoid this bottleneck, we introduce a set of checks based on the notion of *static* or *control-state* reachability.

Let $s = (c_1, \dots, c_n)$ and $s' = (c'_1, \dots, c'_n)$ be two global control states, where for each $i \in [1..n]$, the local path x^i of T_i leads from location c_i to c'_i . We say that s' is *statically* reachable from s if and only if there exists an interleaving of x^1, \dots, x^n that obeys the scheduling constraints imposed by the locks while ignoring data (which is the consistency between shared variable accesses).

Definition 2 (Static Visible Successor). For global control states s, s' in program \mathcal{P} , we say that s' is a static visible successor of s iff

- s is statically reachable from s via some interleaved computation, and
- along this computation, at most one global operation is present.

Here the second condition ensures that s' can be immediately reached from s (hence a successor). Let $\text{Succ}_{\mathcal{P}}(s)$ be the set of static visible successors of s in program \mathcal{P} . Our static lock removal strategy is stated as follows.

Theorem 1 (Behavior Preservation). Let program \mathcal{P}' result from program \mathcal{P} via lock removal. If for each global control state s of \mathcal{P} , we have $\text{Succ}_{\mathcal{P}}(s) = \text{Succ}_{\mathcal{P}'}(s)$, then the two programs have the same behavior as defined by the partial orders of global operations.

Intuitively, if no new global control state becomes reachable from the initial state, then there is certainly no new program behavior. For brevity, we omit the proof. A crucial property of Theorem I is that the static reachability check can be turned into a conceptual lock removal procedure as follows:

1. Enumerate the set S of global control states of the given trace program.
2. For each $s \in S$, compute the set $\text{Succ}_{\mathcal{P}}(s)$ of static visible successors.
3. For each lock statement $lk\text{-stmt}$ in thread T_i , if there exists a global control location s such that, removing $lk\text{-stmt}$ would add a new successor s' that is not in $\text{Succ}_{\mathcal{P}}(s)$, we must retain $lk\text{-stmt}$ else $lk\text{-stmt}$ is removed.

There are two remaining problems. First, given two global control states s, s' , how to efficiently decide whether s' is a static visible successor of s . Second, how to efficiently compute the set of static visible successors of s while avoiding the naive enumeration of all global control states. We will address these two problems in the next section.

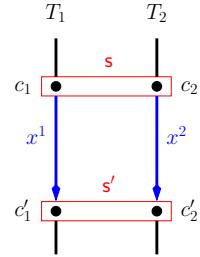
4 Compositional Lock Removal

We present a compositional analysis for static lock removal to avoid the exponential blowup incurred by naively enumerating the global control states. Our method is thread-modular in that the lock removal computation involves only thread-local reasoning, and therefore has a linear worst-case time complexity in the program size.

4.1 Deciding Static Reachability

We leverage an existing procedure [18] to decide the static reachability between two global control states. The procedure is both sound and complete for 2-threaded programs with nested locks. For programs with more than two threads, the procedure remains sound but is not complete. This is acceptable because, as long as it shows that s' is statically *unreachable* from s , the unreachability is guaranteed to hold.

The procedure in [18] can be viewed as a generalization of the standard *lockset* analysis [24]. The key insight is that, to decide whether $s' = (c'_1, c'_2)$ is statically reachable from $s = (c_1, c_2)$, for example, in a 2-threaded program, merely checking the disjointness of the set of locks held by T_1 and T_2 at c'_1 and c'_2 is not enough (see the figure on the right). Although overlapping locksets prove that s' is not reachable from s , the disjointness of the locksets is not sufficient to prove that s' is reachable from s . Instead, reachability can be decided more accurately by first computing a *lock access pattern (LAP)* for each path from c_i to c'_i , where $i \in [1..2]$, and then checking whether the LAPs are consistent.



Definition 3 (Lock Access Pattern). The lock access pattern for path x^i from c_i to c'_i in thread T_i , denoted $\text{LAP}(c_i, c'_i)$, is a tuple $(L_1, L_2, \text{bah}, \text{fah}, \text{Held}, \text{Acq})$ where

- L_1 and L_2 are the set of locks held by T_i at c_i and c'_i , respectively;
- bah and fah are the backward and forward acquisition histories, respectively:
 - for each lock $l \in L_2$ held at c'_i , $\text{bah}(l)$ is the set of locks acquired (and possibly released) after the last acquisition of l along path x^i from c_i to c'_i ;
 - for each lock $l \in L_1$ held at c_i , $\text{fah}(l)$ is the set of locks released (and possibly acquired) since the last release of l in traversing x^i backward from c'_i to c_i .
- Held is the set of locks that are held in every state along path x^i from c_i to c'_i ;
- Acq is the set of locks that are acquired (and possibly released) along path x^i .

A key feature of this LAP-based static analysis procedure is that all computations are local to each individual thread, which is crucial in ensuring scalability.

Decomposition Result. The static reachability from s to s' can be decided by checking whether the corresponding lock access patterns are consistent. For ease of exposition, we present the result for programs with two threads. However, the result, as well as all the other subsequent results, is applicable to programs with n threads.

Let $s = (c_1, c_2)$ and $s' = (c'_1, c'_2)$ be two global control states, and $\text{LAP}(c_1, c'_1) = (L_1^1, L_2^1, \text{bah}^1, \text{fah}^1, \text{Held}^1, \text{Acq}^1)$ and $\text{LAP}(c_2, c'_2) = (L_1^2, L_2^2,$

$\text{bah}^2, \text{fah}^2, \text{Held}^2, \text{Acq}^2$) be the lock access patterns. Then s' is statically reachable from s iff

1. $L_1^1 \cap L_2^2 = \emptyset$, and $L_2^1 \cap L_1^2 = \emptyset$;
2. there do not exist locks $l \in L_1^1$ and $l' \in L_2^2$ such that $l \in \text{fah}^2(l')$ and $l' \in \text{fah}^1(l)$;
3. there do not exist locks $l \in L_2^1$ and $l' \in L_1^2$ such that $l \in \text{bah}^2(l')$ and $l' \in \text{bah}^1(l)$;
4. $\text{Acq}^1 \cap \text{Held}^2 = \emptyset$, and $\text{Acq}^2 \cap \text{Held}^1 = \emptyset$.

For n -threaded programs, the only significant difference would be in conditions 2 and 3, wherein one has to account for the cases in which n threads form a cyclic dependency that may span multiple threads instead of just two.

4.2 Compositional Analysis

To avoid the expensive enumeration of global control states as described in Theorem 1, we compute for each individual thread, all pairs of local control states that may corresponds to some static visible successors. More specifically, a pair (c_i, c'_i) of control locations in thread T_i is called a *pair of interest (POI)* iff

- c_i and c'_i correspond to either shared variable accesses or lock acquisitions, and
- there exists a local path x^i in T_i from c_i to c'_i such that no other shared variable access or lock acquisition occurs between c_i and c'_i .

Our compositional lock removal procedure is given in Algorithm 1. After computing the POIs of each thread T_i , it traverses that thread to collect the lock access patterns for all POIs. Let LP_i denote the set of all lock access patterns in T_i . Note that LP_i can be computed via a single traversal pass of thread T_i (step 4).

Algorithm 1. Compositional Lock Removal

```

1: Input: Threads  $T_1, T_2$ 
2: for each thread  $T_i$  do
3:   Enumerate all pairs of interest  $\text{POI}(T_i)$ .
4:   Traverse the local path in  $T_i$  to compute  $\text{LAP}(c_i, c'_i)$  for each pair  $(c_i, c'_i) \in \text{POI}(T_i)$ .
5:   Let  $\text{LP}_i$  be the set of lock access patterns of all POIs in  $T_i$ .
6: end for
7: for each pair  $(lap_1, lap_2)$  where  $lap_i \in \text{LP}_i$  for all thread index  $i \in [1..2]$  do
8:   if  $lap_1, lap_2$  are inconsistent then
9:     Identify the set of lock statements that are the root causes of inconsistency.
10:    end if
11: end for
12: Remove lock statements that are not the root causes of inconsistency for any pair.

```

Instead of iterating through the set of all global control states, Algorithm 1 considers all pairs (lap_1, lap_2) of lock access patterns that are inconsistent (step 7). Note that lap_i corresponds to some pair $(c_i, c'_i) \in \text{POI}(T_i)$ and the inconsistency of lap_1 and lap_2 means that there exist some lock statements that prevent (c_1, c_2) from reaching

(c'_1, c'_2) . In this case, we need to identify a minimum subset of lock statements that are sufficient to establish this inconsistency, and retain these lock statements. Finally any lock statement that is not responsible for causing an inconsistency between any pair of lock access patterns does not impact the reachability between any pair of global control states, and is therefore removed.

It is worth pointing out that the lock statements (to be retained) can be identified from the lock access patterns (lap_1 and lap_2) alone, without considering the global control states or the POIs that generate these lock access patterns. In other words, we can implicitly isolate the set of non-reachable pairs of global control states without explicitly enumerating them. The algorithm can also be extended to programs with n threads, by changing step 7 to check for inconsistent tuples of the form (lap_1, \dots, lap_n) , as opposed to the inconsistent pair (lap_1, lap_2) .

4.3 Identifying the Locks to Be Retained

If s' is not statically reachable from s in the original program \mathcal{P} , according to Section 4.1, at least one of the conditions in the decomposition result must be violated. From these conditions, we can isolate the root causes that prevent s from reaching s' statically. Our observation is that if s' is not statically reachable from s in \mathcal{P} , then we need to make sure that s' is not reachable from s in the transformed program \mathcal{P}' . The behavior preservation can be guaranteed if we retain at least some (but not all) of the lock statements that prevent s from reaching s' .

Given an inconsistent pair lap_1 and lap_2 of lock access patterns, we can define a reachability barrier by isolating the locks causing the inconsistency. To this end, for each pair (s, s') of global control states where $s = (c_1, c_2)$ and $s' = (c'_1, c'_2)$, we define a *reachability barrier*, denoted $RB(s, s')$, which is the set of all locksets (L) for which at least one of the following conditions holds:

- $L = \{l\}$, where l is held at both c_1 and c_2 or at both c'_1 and c'_2 (violating condition 1 of the decomposition result);
- $L = \{l, l'\}$, where l and l' are held at c_1 and c_2 , respectively, such that $l \in fah(l')$ and $l' \in fah(l)$ (violation of condition 2);
- $L = \{l, l'\}$, where l and l' are held at c'_1 and c'_2 , respectively, such that $l \in bah(l')$ and $l' \in bah(l)$ (violation of condition 3);
- $L = \{l\}$, where l is held throughout x^1 (or x^2) and is acquired along x^2 (or x^1) (violation of condition 4).

Note that in order to ensure that s' remains unreachable from s , it suffices to retain the locks belonging to some lockset in $RB(s, s')$ as that will ensure that at least one condition of the decomposition result is violated.

5 Applying Lock Removal to the Running Example

We now use our new method to remove all locks in the trace program shown in Fig. 1(b) while preserving the program behavior.

We start by identifying the pairs of interest. In the path x^1 shown in Fig. 11(b), there are three lock acquisition statements, i.e. locations $1a$, $5a$ and $6a$, and two shared variable accesses, i.e., $0a$ and $10a$ (the initial state is always treated as a shared variable access). This leads to the pairs of interest $\text{POI}(x^1) = \{(0a, 0a), (0a, 1a), (1a, 1a), (1a, 5a), (5a, 5a), (5a, 6a), (6a, 6a), (6a, 10a)\}$. Similarly, $\text{POI}(x^2) = \{(0b, 0b), (0b, 3b), (3b, 3b), (3b, 4b), (4b, 4b), (4b, 8b), (8b, 8b)\}$.

Next, we compute the lock access patterns generated by all pairs of interest in paths x^1 and x^2 . Toward that end, we compute the lap2POI function for x^2 that maps each lock access pattern lap that is encountered to the set of POIs of x^2 that generate that pattern. For each (c_2, c'_2) in the set $\{(0b, 0b), (0b, 3b), (3b, 3b), (8b, 8b)\}$, no lock is held at either c_2 or c'_2 and no lock is acquired along the sub-sequence of x^2 from c_2 to c'_2 . Thus all the entries in the lock access pattern tuples for these pairs are empty (note that if a thread is at location $3b$ it means that the statement at $3b$ hasn't been executed yet, i.e., lock held at location $3b$ is \emptyset).

Consider now the pair of interest $(4b, 8b)$. We show that $\text{LAP}(4b, 8b) = (\{\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$. The first two entries in the tuple are the locksets held at $4b$ and $8b$ which are $\{A\}$ and \emptyset , respectively. Since no lock is held at the final state $8b$, the forward acquisition histories, i.e., the fourth entry of the tuple is empty. On the other hand, lock A is held at the initial state $4b$. This lock is released at $7b$. However before it is released T_2 also releases B at $6b$. Thus B is in the backward acquisition history of A which is reflect in the third entry of the tuple. Also, since lock B is acquired at location $4b$, we have $\text{Acq} = \{B\}$ (6th entry). Finally, since there exists no lock that is held at all states, we have $\text{Held} = \emptyset$ (5th entry). Similarly, we may compute the lock access patterns for the remaining pairs of interest (see Fig. 3(b)). Similarly, we compute the lap2POI function for x^1 (see Fig. 3(a)).

From Fig. 3(a) and 3(b), we compute the inconsistent pairs (p_1, p_2) of lock access patterns where

1. $p_1 = (\{\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$, $p_2 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$: Held and Acq fields of p_1 and p_2 , respectively, have the common lock A .
2. $p_1 = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{A\})$ and $p_2 = (\{\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: Acq and Held fields of p_1 and p_2 , respectively, have the common lock A .
3. $p_1 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$ and $p_2 = (\{\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: Acq and Held fields of p_1 and p_2 , respectively, have the common lock A .
4. $p_1 = (\{\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$ and $p_2 = (\{\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: L_1 fields have the common lock A .
5. $p_1 = (\{\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$ and $p_2 = (\{\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$: L_1 fields have the common lock A .
6. $p_1 = (\{\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$ and $p_2 = (\{\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset)$: L_1 fields have the common lock A .
7. $p_1 = (\{\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$ and $p_2 = (\{\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\})$: L_1 fields have the common lock A .
8. $p_1 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$ and $p_2 = (\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\})$: L_2 fields have the common lock A .

Note that in each of the above cases, the only lock occurring in the reachability barriers of the non-reachable pairs of global control states is A . Since lock B does not occur

in any of the reachability barriers, in the first iteration, we can remove all statements locking/unlocking B .

Now we repeat the lock removal procedure again on the trace program in Fig. 1(b), by converting statements 6a, 8a, 4b and 6b to `nop`. These new traces generate the `lap2POI` functions shown in Figs. 3(c) and (d). Note that now all pairs of access patterns are mutually consistent. Thus the reachability barriers for all pairs of global control states are empty. Hence all locks in the original traces can now be removed giving us the traces with no lock statements.

$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \rightarrow \{(0a, 0a), (0a, 1a), (1a, 1a), (5a, 5a), (10a, 10a)\}$ $(\emptyset, \emptyset, \emptyset, \emptyset, \{A\}) \rightarrow \{(1a, 5a)\}$ $(\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\}) \rightarrow \{(5a, 6a)\}$ $(\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset) \rightarrow \{(6a, 6a)\}$ $(\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\}) \rightarrow \{(6a, 10a)\}$
(a)
$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \rightarrow \{(0b, 0b), (0b, 3b), (3b, 3b), (8b, 8b)\}$ $(\emptyset, \{A\}, \emptyset, \{(A, \{\})\}, \emptyset, \{A\}) \rightarrow \{(3b, 4b)\}$ $(\{A\}, \{A\}, \emptyset, \emptyset, \{A\}, \emptyset) \rightarrow \{(4b, 4b)\}$ $(\{A\}, \emptyset, \{(A, \{B\})\}, \emptyset, \emptyset, \{B\}) \rightarrow \{(4b, 8b)\}$
(b)
$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \rightarrow \{(0a, 0a), (0a, 1a), (1a, 1a), (5a, 5a), (10a, 10a)\}$ $(\emptyset, \emptyset, \emptyset, \emptyset, \{A\}) \rightarrow \{(1a, 5a), (5a, 10a)\}$
(c)
$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \rightarrow \{(0b, 0b), (0b, 3b), (3b, 3b), (8b, 8b)\}$ $(\emptyset, \emptyset, \emptyset, \emptyset, \{A\}) \rightarrow \{(3b, 8b)\}$
(d)

Fig. 3. The `lap2POI` function for x^1 (left) and x^2 (right)

Generalizations. So far, for ease of exposition, we have presented all the algorithms using concurrent trace programs with two threads. However, our results can be extended to programs with an arbitrary but fixed number of threads. This generalizations do not require additional insights. The only difference from the 2-thread case is that we need an efficient technique to decide static reachability between global control states which are now n -tuples of the form (c_1, \dots, c_n) , where each c_i is either a shared variable access or a lock acquisition in thread T_i . This is achieved via a straightforward extension of the decomposition result in Section 4.1. That is, for each pair of threads, we check whether their lock access patterns (LAPs) are consistent.

So far we have discussed only mutex locks. A typical real-world concurrent program in Java or C (with POSIX threads) may have additional concurrency primitives such as thread creation and join operations, wait/notify/notifyall, as well as reentrant locks. The presence of these synchronization primitives does not affect the soundness of our lock removal algorithm. The reason is that, if s' is statically unreachable from s according to locks (while ignoring data and other concurrency primitives), it is guaranteed to be unreachable when more synchronization constraints are considered. At the same time, if there is a way to incorporate the causality constraints imposed by other concurrency primitives, one can more accurately determine the reachability between global control

states, therefore leading to the identification and removal of potentially more redundant lock statements. To this end, we have incorporated the universal causality graph based analysis in [19] during our implementation of the proposed lock removal method. However, we note that this UCG-based analysis is orthogonal to lock removal, and can be carried out once in the beginning of the computation.

6 Experiments

We have evaluated the lock removal method in the context of an SMT-based runtime predictive analysis [28][29], to quickly remove the lock statements that are redundant and therefore ease the burden of modeling and checking by the SMT solvers.

We now provide a brief overview of the symbolic predictive analysis. Given a multi-threaded Java or C program and a user-defined test case, the predictive analysis procedure first instruments the program code to add self-logging capability, and then uses stress tests to detect concurrency failures. However, due to the scheduling nondeterminism and the astronomically large number of interleavings, it is often difficult to uncover the concurrency bugs. If testing fails to detect any bug, we start a post-mortem analysis of the logged execution trace.

In this subsequent analysis, first we use a simple control flow analysis to compute the *potential bugs*. Consider the one-variable three-access atomicity violation [21][11] as an example. In this case, a potential bug is a sequence $t_c \dots t_r \dots t_{c'}$ of program statements such that: (1) t_c and $t_{c'}$ are intended to be executed atomically by one thread, (2) t_r is in another thread and is data dependent with both t_c and $t_{c'}$. Then we use a more precise static analysis based on the *universal causality graph* (UCG [19]) to prune away the obviously bogus violations.

For each remaining potential violation, we call the SMT-based symbolic procedure to decide if there exists a valid interleaving under which the violation is feasible. In this context, an interleaving is feasible if it satisfies both the synchronization consistency (e.g. locks) and the shared memory consistency. Please refer to [28][29][26] for more information about the symbolic encoding. Here we assume the sequential consistency (SC) memory model. We have used the YICES solver from SRI [8] in our experiments. Since having more lock statements generally leads to more logical constraints and therefore a higher cost for SMT solving, we have used lock removal before the SMT-based analysis, to remove the redundant lock statements.

We conducted experiments using the following benchmarks¹. The Java programs come from various public benchmarks [16][17][15][27]. The C programs are the PThreads implementation of two sets of known bug patterns. The first set (*At*) mimics an atomicity violation in the Apache web server code (c.f. [21]), where *At1* is the original program, while *At1a* and *At2a* are generated by adding code to the original programs to remove the atomicity violations. The second set (*bank*) is a parameterized version of the *bank* example [10], where the original program *bank-av* has a well-known atomicity violation and the remaining two are various attempts of fixing it. All our experiments were conducted on a PC with 1.6 GHz Intel processor and 2GB memory running Fedora.

¹ The benchmarks are available at <http://www.nec-labs.com/~chaowang/pubDOC/LnW.tar.gz>

Table I shows the results. The first five columns show the statistics of the trace program, including the name, the number of threads, the total number of events, the number of lock/unlock events, and the number of named locks. The next nine columns show the statistics of the lock removal computation. In particular, Columns 6-9 show the total number of pairs of interest (POI), the number of POIs without any held lock (POI-e), the number of POIs with non-trivial lock acquisition histories (POI-h), and the maximum nesting depths of locks (max-h). The fact that *max-h* is often zero helps to make our analysis scale to real-life programs. Columns 10-11 show the total number of relevant pairs of global control states, and the number of pairs wherein one state is unreachable from the other. Columns 12 and 13 show the number of critical sections (pairs of lock-unlock statements) in the original and transformed programs, respectively. Column 14 shows the total time (in seconds) taken for the lock removal computation.

Table 1. Results: Using lock removal to improve symbolic analysis. *mem* means memory-out.

Concurrent Trace Program				Lock Removal Computation								Symbolic Analysis					
name	thrd	events	lk-evs	lk-v	POI	POI-e	POI-h	max-h	vis-ne	vis-ch	lk-r	rm-r	time(s)	p-avs	r-avs	pre(s)	post(s)
ra.Main	3	55	12	3	23	7	0	0	65	0	5	3	0.0	2	0	0.0	0.0
connect	4	97	16	1	43	29	0	0	1526	0	8	0	0.0	6	0	0.1	0.1
hedcex	1	122	35	7	1	0	0	0	0	0	0	0	0.0	0	0	0.0	0.0
liveness	7	283	44	9	105	68	0	0	10272	0	15	0	0.2	36	0	0.4	0.4
BarrierB1	10	653	108	2	307	168	0	0	69498	0	35	14	0.9	102	0	10.5	3.0
BarrierB2	13	805	136	2	409	217	0	0	120659	0	49	21	1.6	87	0	54.5	7.4
account1	11	902	146	21	230	134	0	0	43690	0	72	30	0.7	140	2	1.8	0.9
philo	6	1141	126	6	433	260	0	0	147294	0	63	10	2.2	81	0	42.5	19.4
account2	21	1747	282	41	442	260	0	0	171400	0	140	60	2.6	280	3	8.7	2.4
Daisy1	3	2998	422	10	843	105	29	1	17249	141	204	175	0.3	7	0	mem	21.3
Elevator1	4	3004	370	11	893	28	0	0	1453	0	184	174	0.1	4	0	29.6	0.7
Elevator2	4	5001	610	11	1992	116	0	0	25435	0	304	257	0.7	8	0	mem	4.3
Elevator3	4	8004	1128	11	2369	214	0	0	81890	0	563	468	1.9	12	0	mem	28.2
Tsp	4	45653	20	5	87	4	0	0	20	0	8	6	0.0	0	0	0.0	0.0
At1	3	88	6	1	14	7	0	0	60	0	3	0	0.0	3	0	1.0	0.0
At1a	3	100	8	1	17	10	0	0	126	0	4	0	0.0	4	0	1.0	0.0
At2a	3	462	126	2	156	149	32	1	38208	9216	52	16	0.6	52	16	2.0	0.6
Bank-av	3	748	20	3	160	104	0	0	28776	0	40	8	0.4	40	8	8.0	0.4
Bank-sav	3	852	28	3	195	139	0	0	51510	0	56	8	0.7	56	8	8.0	0.7
Bank-fix	3	856	32	3	204	147	16	1	57612	12540	64	8	0.8	64	8	9.0	0.8

Finally, the last four columns in Table I show the impact of lock removal on the performance of a runtime verification procedure. Recall that, for each of the potential atomicity violations, we use symbolic analysis to decide whether it is a real atomicity violation. Here we first show the total number of potential atomicity violations (p-avs) that are collected by a simple static analysis, and then show the number of real atomicity violations found by the precise symbolic analysis (r-avs). Please refer to [29][19] for more details on predicting atomicity violation. The last two columns compare the runtime of symbolic analysis with and without lock removal. The results clearly show that lock removal has made the predictive verification step more efficient. Note that for *Daisy1* (which is file system) and *Elevator2*, without lock removal, symbolic execution would run out of the 2GB memory limit, whereas after lock removal, they were able to finish in short time.

7 Related Work

Existing work on automatically removing unnecessary synchronizations has concentrated mostly on performance optimization and on eliminating thread-local locks [3][4][6][30], i.e. locks that have been acquired or released by a single thread or used to protect an object accessed by a single thread. The difference among these methods lies in how they identify shared/escaped objects. For example, Blanchet [3] uses a flow-insensitive escape analysis both to allocate thread-local objects on the stack and to eliminate synchronization from stack-allocated objects. Bogda et. al. [4] also use a flow-insensitive escape analysis to eliminate synchronization from thread local objects, but the analysis is limited to thread-local objects that are only reachable by paths of one or two references from the stack. Choi et al. [6] perform an inter-procedural points-to analysis to classify objects as globally escaping, escaping via an argument, and not escaping. When synchronizing, the compiler eliminates synchronizations for thread-local objects, while preserving Java semantics by flushing the local processor cache.

Ruf [22] combines a thread behavior analysis with a unification based alias analysis to removal unnecessary synchronizations. Aldrich et al. [1] propose three analysis to optimize the synchronization opportunities: lock analysis, unshared field analysis, and multithreaded object analysis. Lock analysis computes a description of the monitors held at each synchronization point so that reentrant locks and enclosed locks can be eliminated. Unshared field analysis identifies unshared fields so that lock analysis can safely identify enclosed locks. Finally, multithreaded object analysis identifies which objects may be accessible by more than one thread. This enables the elimination of all synchronization on objects that are not multi-threaded. Zee and Rinard [30] present a static program analysis for removing unnecessary write barriers in Java programs that use generational garbage collection.

In contrast, the focus of our work is not to identify which objects are *effectively thread-local*, which objects are shared, or when they are shared, by multiple threads, but to identify more optimization opportunities on the truly shared objects and yet redundant locks. To the best of our knowledge, this is the first such lock removal algorithm. It is generally applicable, based on a rigorous and unified concurrency analysis framework. It is also practically efficient, due to the use of lock access patterns, which involves only thread-local computation.

In the formulation of our efficient check for behavior preservation, we have leveraged the lock access patterns [18], since our trace program has a fixed number of threads interacting with only nested locks. To extend the method from trace programs to whole programs, one might need to leverage the more advanced machinery in [13][9] to deal with locks interacting with dynamic thread creation.

In the literature, there has also been some work on reducing the run-time cost of synchronizations, e.g. by making their implementation more efficient (e.g. [2]) rather than removing the unnecessary ones. These techniques complement ours. Our local removal algorithm is also different from lock coarsening [7], which optimizes the necessary synchronizations, e.g. those arising from acquiring and releasing a lock multiple times in succession. Converting multiple lock operations into one, in general, changes the program behavior, and therefore one must take care not to introduce deadlock.

8 Conclusions

In this paper, we have presented an efficient and fully automatic lock removal technique for concurrent trace programs. A key feature of our method is that it is compositional in nature, i.e., hinges on a thread local analysis, which makes it applicable to large, realistic programs. Furthermore, our technique guarantees the preservation of program behaviors, i.e., partial orders induced on shared variable accesses. These features make it a standalone utility with many wide ranging applications, including performance optimization as well as improving the efficacy of concurrent program analysis like runtime verification, model checking and dataflow analysis. As a concrete application, we demonstrated the use of our lock removal technique in enhancing the scalability of predictive analysis in the context of runtime verification of concurrent programs.

References

1. Aldrich, J., Chambers, C., Sirer, E.G., Eggers, S.J.: Static analyses for eliminating unnecessary synchronization from Java programs. In: International Symposium on Static Analysis, pp. 19–38 (1999)
2. Bacon, D.F., Konuru, R.B., Murthy, C., Serrano, M.J.: Thin Locks: Featherweight synchronization for Java. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 258–268 (1998)
3. Blanchet, B.: Escape analysis for object-oriented languages: Application to Java. In: ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 20–34 (1999)
4. Bogda, J., Hölzle, U.: Removing unnecessary synchronization in Java. In: ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 35–46 (1999)
5. Chen, F., Roșu, G.: Parametric and Sliced Causality. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 240–253. Springer, Heidelberg (2007)
6. Choi, J.-D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for Java using escape analysis. ACM Trans. Program. Lang. Syst. 25(6), 876–910 (2003)
7. Diniz, P.C., Rinard, M.C.: Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. J. Parallel Distrib. Comput. 49(2), 218–244 (1998)
8. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
9. Esparza, J., Ganty, P.: Complexity of pattern-based verification for multithreaded programs. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 499–510 (2011)
10. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Parallel and Distributed Processing Symposium, p. 286 (2003)
11. Farzan, A., Madhusudan, P., Sorrentino, F.: Meta-analysis for Atomicity Violations under Nested Locking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 248–262. Springer, Heidelberg (2009)
12. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: Parallel and Distributed Processing Symposium (2004)
13. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-Lock-Sensitive Forward Reachability Analysis for Concurrent Programs with Dynamic Process Creation. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 199–213. Springer, Heidelberg (2011)

14. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Springer (1996)
15. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Software Tools for Technology Transfer 2(4) (2000)
16. Joint cav/issta special even on specification, verification, and testing of concurrent software, <http://research.microsoft.com/qadeer/cavissta.htm>
17. The java grande forum benchmark suite, http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html
18. Kahlon, V.: Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks. In: Symposium on Logic in Computer Science, pp. 27–36 (2009)
19. Kahlon, V., Wang, C.: Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 434–449. Springer, Heidelberg (2010)
20. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
21. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Architectural Support for Programming Languages and Operating Systems, pp. 37–48 (2006)
22. Ruf, E.: Effective synchronization removal for Java. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 208–218 (2000)
23. Sadowski, C., Freund, S.N., Flanagan, C.: SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 394–409. Springer, Heidelberg (2009)
24. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
25. Sen, K., Roşu, G., Agha, G.: Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 211–226. Springer, Heidelberg (2005)
26. Sinha, N., Wang, C.: On interference abstractions. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 423–434 (2011)
27. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. Object Technology 3(6) (2004)
28. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic Predictive Analysis for Concurrent Programs. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009)
29. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-Based Symbolic Analysis for Atomicity Violations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)
30. Zee, K., Rinard, M.C.: Write barrier removal by static analysis. In: ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 191–210 (2002)

How to Prove Algorithms Linearisable

Gerhard Schellhorn¹, Heike Wehrheim², and John Derrick³

¹ Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de

² Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

³ Department of Computing, University of Sheffield, Sheffield, UK
J.Derrick@dcs.shef.ac.uk

Abstract. Linearisability is the standard correctness criterion for concurrent data structures. In this paper, we present a *sound* and *complete* proof technique for linearisability based on backward simulations. We exemplify this technique by a linearisability proof of the queue algorithm presented in Herlihy and Wing’s landmark paper. Except for the manual proof by them, none of the many other current approaches to checking linearisability has successfully treated this intricate example. Our approach is grounded on *complete mechanisation*: the proof obligations for the queue are verified using the interactive prover KIV, and so is the general soundness and completeness result for our proof technique.

1 Introduction

The advent of multi- and many-core processors will see an increased usage of concurrent data structures. These are implementations of data structures like queues, stacks or hashtables which allow for concurrent access by many processes at the same time. Libraries such as `java.util.concurrent` offer a vast number of such concurrent data structures. To increase concurrency, these algorithms often completely dispose with locking, or only lock small parts of the structure. This inevitably leads to race conditions. Indeed, the designers of such algorithms do not aim at race-free but at *linearisable* algorithms. Linearisability [14] requires that fine-grained implementations of access operations (e.g., insertion or removal of an element) appear as though they take effect “instantaneously at some point in time” [14], thereby achieving the same effect as an atomic operation.

Recently, a number of new approaches to proving linearisability have appeared, some supported by theorem provers (like our own), some automatic based on user-annotated algorithms and some manual (see Section 7). Looking at these approaches, one finds that a number of techniques (including our own so far) get adapted every time a new type of algorithm is treated. Every new “trick” designers build into their algorithms to increase performance (e.g., like a mutual push and pop elimination for stacks, or lazy techniques) requires an extension of the verification approach.

In this paper, we propose a proof technique which can be used to prove linearisability of *every* linearisable algorithm: Our method is *sound and complete* for linearisability.

The approach is based on *backward simulations* - a technique borrowed from data refinement. More precisely, we show that a fine-grained implementation is linearisable with respect to an abstract atomic specification of the data structure if and only if there is a backward simulation between the specification and the implementation. The use of simulations for showing linearisability is not new; however, current refinement-based approaches (e.g. [9]) are based on both backward *and* forward simulations. We exemplify our approach on the queue implementation of Herlihy and Wing [14]. None of the current other works on linearisability have treated this algorithm; and it is also not clear whether the many approaches tailored towards heap usage (like separation logic or shape analysis based techniques) can successfully verify the queue, as the complexity in the interaction between concurrent processes in the queue is not due to a shared heap (there is no heap involved at all). Along with this queue example we also show how to systematically construct the backward simulations needed in the linearisability proofs.

Last but not least we have a complete mechanisation of our approach. It is complete in the sense that we both carry out the backward simulation proofs for our examples (here, the queue) with an interactive prover (which is KIV [23]), *and* have verified within KIV that the general soundness and completeness proof of our technique is correct. In summary, this paper thus contains three contributions: (1) the proof of soundness and completeness of backward simulations for linearisability, (2) the linearisability proof for the Herlihy and Wing queue, and (3) the full mechanisation of both the example and the general theory.

The next section gives the algorithms for the example. Section 3 defines linearisability as a specific form of refinement. Section 4 gives our main theorem, that linearisability can always be proven with a backward simulation, and Section 5 derives one for the example, showing that this can be done systematically. Section 6 gives some information on the KIV prover, and sketches how the proof obligations for backward simulation could be verified. Full details of all proofs are online [17]. Section 7 gives related work and Section 8 discusses possible improvements and concludes.

2 Example

The queue of [14], which serves as our running example, is a data structure with two operations: *enqueue* appends new elements (of some type T) to the end of the queue and *dequeue* removes elements from the front of the queue. The implementation of the queue uses a shared array AR of unbounded length. All slots of the array are initialised with a value *null*, signalling ‘no element present’. A back pointer *back* into the array stores the current upper end of the array where elements are enqueued. Dequeues operate on the lower end of the array. The pseudocode of the queue operations is as follows:

E0 <i>enq(lv : T)</i>	D0 <i>deq() : T</i>
E1 /* increment */	D1 <i>lback := back; k := 0; lv := null;</i>
(<i>k, back</i>) :=	D2 <i>if k < lback goto D3 else goto D1;</i>
(<i>back, back+1</i>);	D3 (<i>lv, AR[k]</i>) := (<i>AR[k], lv</i>); /* swap */
E2 /* store */	D4 <i>if lv ≠ null then goto D6 else goto D5;</i>
<i>AR[k] := lv;</i>	D5 <i>k := k + 1; goto D2;</i>
E3 <i>return</i>	D6 <i>return(lv)</i>

The *enq* operation simply gets a local copy of *back*, increments *back* (these two steps are executed as an atomic “fetch & increment”) and then stores the element to be enqueued in the array.

The *deq* operation proceeds in several steps: first, it gets a local copy of *back* and initialises a counter *k* and a local variable, *lv*, which is used to store the dequeued element. It then walks through the array trying to find an element to be dequeued. Steps D2 and D5 of the code are a loop consecutively visiting the array elements. At every position *k* visited, the array contents $AR[k]$ is swapped with variable *lv* (i.e., the assignment at D3 is executed in parallel). If the dequeue finds a proper non-*null* element this way ($lv \neq \text{null}$), this will be returned, otherwise the search is continued. In the case where no element can be found in the entire array, *deq* restarts the search. Note that if no *enq* operations occur, *deq* will thus run forever.

The complete specification consists of a number of processes $p \in P$, each capable of executing its queue operations on the shared data structure. For the concrete implementation, therefore, these two algorithms can be executed concurrently by any number of processes - where the individual steps (i.e., the statements in locations E0 to D6) in the operations are taken to be atomic, but crucially can be interleaved. That is, a process may start an *enq* operation (say doing E0 and E1) but then another process may execute its own atomic step (e.g., start a *deq*). Verification that the concrete implementation is somehow correct with respect to abstract, atomic enqueue and dequeue operations is the crux of the problem and linearisability is the proof obligation.

Our proof of linearisability proceeds by showing that the concurrent implementation is a backward simulation of an atomic abstract specification of the queue, i.e. that every step of the implementation can be simulated by the abstract specification in a backward fashion. To this end, we phrase both abstract specification and implementation in terms of *data types*. A data type consists of a state *State* (set of variables) and operations on the state $Op \subseteq State \times State$ (e.g. enqueue or dequeue, or operations like D1, D2, ...). In addition, an initialisation operation $Init : State$ specifies constraints on the initial state and a finalisation operation $Fin \subseteq State \times F$ relates states to global result values from some set *F*. Intuitively, *Fin* fixes those parts of the state that we are interested in and want to observe when comparing the data types. A data type is written as

$$(State, Init, (Op_{p,i})_{p \in P, i \in I}, Fin)$$

Note that we have incorporated processes in here. We take a relational view on operations, and use \circ for composition of relations. Primed variables in operations refer to the after state. Sequences of operations are written as Op^* .

For the abstract queue (omitting *Fin* for the moment), we for instance have $A = (AState, AInit, Enq_{p \in P}, Deq_{p \in P})$ given by

$$\begin{array}{ll} AState \doteq [q : \text{seq } T] & Enq_p(x? : T) \doteq [q' = q \wedge \langle x? \rangle] \\ AInit \doteq [q = \langle \rangle] & Deq_p(x! : T) \doteq [x! = \text{first}(q) \wedge q' = \text{rest}(q)] \end{array}$$

Here, the variable *x?* is an input to and *x!* an output of the operation.

The data type $C = (CState, CInit, (COp_{p,j})_{p \in P, j \in J})$ for the concurrent implementation is more complex. The state consists of the two global variables *back* : \mathbf{N} and

$AR : \mathbb{N} \rightarrow T$ to represent the array with elements of type T . Additionally, the local variables of processes are part of the state, e.g. $lback : P \rightarrow \mathbb{N}$ represents the values of the local variable $lback$ for all the processes. Finally, $pc : P \rightarrow \{N, E1, E2, E3, D1, \dots, D6\}$ defines a program counter for all processes, $pc(p) = N$ means that process p is currently running no operation. Initial states $CInit : CState$ have an empty array, $back = 0$ and $pc(p) = N$ for all $p \in P$.

The operations of this concrete data type are made up of the steps of the algorithm: every line in the algorithm becomes one operation¹. We thus for instance have an operation called $enq1_p$ (line E1 in the enqueue of process p) which is specified as

$$enq1_p \hat{=} [pc(p) = E1 \wedge pc'(p) = E2 \wedge k'(p) = back \wedge back' = back + 1]$$

Here, we use the convention of not mentioning variables of the state which remain unchanged. In a similar way we can define operations for all other steps of the algorithm. Our goal for the next section is to show that all concurrent runs of the algorithm given here faithfully implement queue operations. E.g., a concrete run might start with the sequence $enq0_3 \circ enq1_3 \circ enq0_1 \circ deq0_2$. Does this represent a possible implementation of an abstract run? Formally, we have to prove linearisability and we will do so by showing that the concurrent implementation is (a particular type of) refinement of the abstract atomic specification.

3 Linearisability and Refinement

Linearisability is defined by comparing *histories* created by the atomic queue operations and those created by the concurrent implementation. Histories are sequences of *invoke* and *return* events of particular operations (out of some index set I) by particular processes $p \in P$ with certain input or output values. For example, a possible history of our queue implementation is

$$h = \langle inv(3, enq, a), inv(1, enq, b), inv(2, deq,), inv(4, enq, c), ret(3, enq,), ret(4, enq,) \rangle$$

In this history, process 3 first *invokes* an enqueue operation with argument a . Next, process 1 invokes an enqueue for element b . While these two processes are running, process 2 starts a dequeue, and process 4 invokes an enqueue of c . At the end, first process 3 *returns* from its enqueue and finally process 4. These histories are thus abstracting the algorithm into just its start and end given by the *invokes* and *returns* of the operations. In a *legal* history, a return event of process p from operation i is always preceded by a *matching* (i.e., corresponding) invoke event with the same p and i , while an invoke event may or may not be followed by a matching return. In the latter case, the operation has not yet finished, and the invoke is a member of the set $pi(h)$ of *pending invokes* of history h . For the given history $pi(h) = \{inv(1, enq, b), inv(2, deq,)\}$. In the following, *Event* denotes the set of all events, and we write *Ret* for the set of all return events.

The first step in our proof technique is to add the history created by the algorithms to the data types: We construct *history enhanced* data types collecting histories. The enhancement we define is not specific to the queue example but applies to all concrete

¹ In the KIV specification of the algorithm we split IF-statements into a true and false case.

and abstract data types for which we want to show linearisability. The history enhanced concrete data type $HC = (HCState, HCInit, (HCOp_{p,j})_{p \in P, j \in J}, HCFin)$ gets $HCState \hat{=} CState \wedge [h : Event^*]$. As in the example above, the invoking steps of operations op of processes p (like enq_0_p and deq_0_p for the queue corresponding to lines $E0$ and $D0$) add an event $inv(p, op, a)$ (where a is the input value of op) to the history. Similarly the returning operations add return events, all others leave the history unchanged. Now we can also define a meaningful finalisation operation: $HCFin \subseteq HCState \times F$ for $F = Event^*$ extracts the collected history by defining $HCFin((cs, h), H)$ iff $H = h$.

On the abstract data type we perform a slightly different form of enhancement which is also motivated by our objective of wanting to prove linearisability. Informally, linearisability means that all histories created by the implementation could also be produced by working with an abstract atomic queue. As Herlihy and Wing formulate it: we want the concurrent implementation to “provide the illusion that each operation ... takes effect instantaneously at some point between its invocation and return”. This point in time is usually called the *linearisation point*. The formal definition given in [14], however, is based on comparing concurrent and sequential histories (where the latter are sequences of matching invocation and return pairs). Already [14] note that this definition is not suitable for proofs, therefore like most related work (see Section 7) we prefer an alternative definition that directly formalises the idea of a linearisation point. In the enhancement of the abstract data type $HA = (HState, HAInit, \{Inv_{p,i}, Lin_{p,i}, Ret_{p,i}\}_{p \in P, i \in I}, HAFin)$ we thus add histories *plus* we also split operations in three: an invocation, a linearisation point and a return.

$$\begin{aligned}
HState &\hat{=} AState \wedge [h : Event^*, R : \mathbb{P} Ret] \\
HAInit &\hat{=} AInit \wedge [h = \langle \rangle \wedge R = \emptyset] \\
Inv_{p,i}(in? : In) &\hat{=} [(\neg \exists i' \in in' \bullet inv(p, i', in') \in pi(h)) \wedge \\
&\quad as' = as \wedge R' = R \wedge h' = h \cap \langle inv(p, i, in?) \rangle] \\
Lin_{p,i} &\hat{=} [\exists in, out \bullet inv(p, i, in) \in pi(h) \wedge (\neg \exists out_2 \bullet ret(p, i, out_2) \in R) \wedge \\
&\quad AOp_{p,i}(in, as, as', out) \wedge h' = h \wedge R' = R \cup \{ret(p, i, out)\}] \\
Ret_{p,i}(out! : Out) &\hat{=} [ret(p, i, out!) \in R \wedge h' = h \cap \langle ret(p, i, out!) \rangle \wedge \\
&\quad R' = R \setminus \{ret(p, i, out!) \} \wedge as' = as] \\
HAFin &\hat{=} [H : Event^* \mid H = h]
\end{aligned}$$

As we see here we do not only add a variable h collecting histories but also a variable R , a set of return events. The role of R is to collect return events for those operations which have already taken effect, i.e., which are past the linearisation point but have not yet returned. Abstract execution of operations now consists of three steps: the invocation operation $Inv_{p,i}$ just adds an invoke event to the history, the linearisation operation $Lin_{p,i}$ changes the state according to the original definition of the operation in A , adds a return event to R (now the effect has taken place) and keeps the history. The return operation $Ret_{p,i}$ adds the return event to the history and – now that it is present in h – has to remove it from R . Finalisation again gives the current history.

Note that HA is concurrent in the sense that operations of processes are interleaved. However, the “effect” operation Lin is still atomic and thus faithfully reflects the original abstract data type. These two abstractions HA and HC can thus be compared.

Definition 1. The reachable states of HA are called possibilities. Writing HAOp for the union of all operations of HA, we define

$$\text{Poss}(as, h, R) \triangleq (\text{HAInit} ; \text{HAOp}^*)(as, h, R)$$

Our definition of possibilities is essentially the same as the one in Herlihy and Wing's paper [14], p. 486f. The notation given there is $(as, P, R) \in \text{Poss}(h)$, with a redundant set P , that contains those invokes in $pi(h)$ with no matching return in R . Axioms S , I , C and R correspond one-to-one to our operations HAInit , Inv , Lin and Ret : the premise and conclusion are the pre- and post-state of the operations (our side conditions guarantee legal histories in conclusions, left implicit in [14]).

Lynch [18], Sec. 13.1.2, gives a similar definition of linearisability using the “canonical wait-free automaton for atomic objects”. States of this automaton are essentially (as, P, R) (P is called inv-buffer), traces of the IO automaton correspond to our history. Theorems 9 and 10 of [14] state that possibilities are equivalent to linearisability:

Theorem 1. An implementation data type C is linearisable with respect to some abstract data type A if and only if for every history h created by C there exists a possibility $\text{Poss}(as, h, R)$.

This theorem seems to be universally accepted, and informal arguments for its validity appear in many papers. However, to relate the results given here to the original definition of linearisability, we have mechanised the proof in KIV. The proof is rather complex. It shows that a forward simulation exists between the type HA given here and the abstract data type we have used in [7] for the original linearisability definition. This provided a hint that backward simulation could be a complete proof procedure for the definition given here.

The theorem gives us the option to prove linearisability by showing the existence of possibilities, which can be viewed as a form of a refinement, given next.

Definition 2. Let $A = (A\text{State}, A\text{Init}, (A\text{Op}_{p,i})_{p \in P, i \in I}, A\text{Fin})$ and $C = (C\text{State}, C\text{Init}, (C\text{Op}_{p,j})_{p \in P, j \in J}, C\text{Fin})$ be abstract and concrete data types respectively.

A program is a sequence of operation (indices) $\text{Prg} = j_1 \dots j_n$; and running a program on the data type C gives the execution

$$\text{Prg}(C) \triangleq C\text{Init} ; C\text{Op}_{j_1} ; \dots ; C\text{Op}_{j_n} ; C\text{Fin}$$

C is a data refinement of A , denoted $\Box C \sqsubseteq A$, if for all programs Prg , $\text{Prg}(C) \subseteq A\text{Init} ; A\text{Op}^* ; A\text{Fin}$. An empty concrete program must refine the empty abstract program.

Note that this is a very weak form of refinement as it assumes that the effect of a particular program in C can be achieved with some arbitrary program ($A\text{Op}^*$) in A . This is crucial for our approach since for complex linearisable algorithms – such as the one we consider in this paper – one concrete step in the implementation may correspond to the execution of several linearisation steps in the abstract data type. This type of refinement applied on the enhanced data types coincides with linearisability.

² Note that the literature on refinement usually writes the notation the other way round.

Theorem 2. $HC \sqsubseteq HA$ iff C linearisable wrt. A .

Proof: The concrete histories h are the values returned by finalisation of HC . Refinement implies that there is an abstract run which also produces h . This run reaches a state (as, h, R) in HA before finalisation, i.e. $Poss(as, H, R)$ holds, and linearisability follows by Theorem I. On the other hand, if linearisability holds, and h is a concrete history, then there is a possibility $Poss(as, h, R)$ by Theorem I, so refinement holds, since finalisation will give h . \square

4 Proving Linearisability with Backward Simulation

Data refinement is the process of adding implementation detail to an initial abstract algorithm, and standard results show that forward and backward simulations are sound and jointly complete for verifying refinements (see [6] for an overview).

The fact that linearisability can be expressed in terms of refinement also underlies the work of Doherty, Groves et al. [9][12]. However, their work as well as many others assume that linearisability needs *both* backward and forward simulation to be complete (and, e.g., [9] uses both). Here, we show that in fact backward simulation alone is already complete for proving linearisability.

Definition 3 (Backward simulation). Let $A = (AState, AInit, (AOp_{p,i})_{p \in P, i \in I}, AFin)$ and $C = (CState, CInit, (COp_{p,j})_{p \in P, j \in J}, CFin)$ be two data types. A relation $BS \subseteq CState \times AState$ is a backward simulation from C to A , denoted $C \preceq_{BS} A$, if the following conditions hold:

- *Initialisation*: $CInit \circ BS \subseteq AInit$,
- *Finalisation*: $CFin \subseteq BS \circ AFin$,
- *Correctness*: $\forall p \in P, j \in J \bullet COp_{p,j} \circ BS \subseteq BS \circ AOp^*$.

The correctness condition is weaker than usual (to match the weak data refinement) in that it only requires a concrete operation to be matched by an arbitrary sequence of abstract operations. Note that BS is often called *abstraction relation*: given a concrete state one has to define what the possible corresponding abstract states are.

The main result of this paper is that backward simulations are sufficient and we can avoid forward simulations entirely when verifying linearisability. The proof relies on the following two observations.

Proposition 1. ([4], p. 487) *Possibilities are prefix-closed: If $Poss(as, h_0 \cap h, R)$ for some histories h_0, h , set of returns R and abstract state as , then there are as_0 and R_0 , such that $Poss(as_0, h_0, R_0)$ and $HAOp^*((as_0, h_0, R_0), (as, h_0 \cap h, R))$.*

Proof: Simple induction over the number of operation executions necessary to reach the final state $(as, h_0 \cap h, R)$, since every operation adds at most one event and we start with the empty history. \square

Proposition 2. *The reachable states (as, h, R) of HA satisfy an invariant called $retsforpis(h, R)$ (returns for pending invokes only) which says that all return events in R have a process p with a corresponding invoke event in $pi(h)$.*

Again the proof is by induction on the number of operation executions. This now lets us formulate and prove our main theorem which shows that backward simulation is sound and complete for linearisability.

Theorem 3. *Let C, A be a concrete and an abstract data type, and HC, HA their history enhancements as defined above. Then $HC \preceq_{BS} HA$ iff C linearisable wrt. A .*

Proof: The easy direction from left to right just combines soundness of backward simulation and Theorem 2. For the other direction, assume C is linearisable wrt. A . We define a relation BS as

$$\begin{aligned} BS((cs, h), (as, H, R)) \hat{=} & h = H \wedge Poss(as, H, R) \\ & \wedge (H = \langle \rangle \Rightarrow AInit(as)) \end{aligned}$$

and prove the three proof obligations which show backward simulation.

- For *Initialisation*, we must prove that $HCInit(cs, h)$ implies $HAInit(as, H, R)$ when BS holds. Since $h = \langle \rangle$, we have $H = h = \langle \rangle$ and $ASInit(as)$. It remains to show that $R = \emptyset$. This follows from Proposition 2, since $pi(\langle \rangle) = \emptyset$.
- *Finalisation* requires to find an abstract (as, H, R) for every (cs, h) , such that BS holds. Since C is linearisable, there is a state (as, h, R) with $Poss(as, h, R)$. If h is nonempty, this state is already sufficient. Otherwise, state (as, h, R) was reached from an initial state $(as_0, h_0, R_0) \in HAInit$ with $h_0 = \langle \rangle$, $R_0 = \emptyset$ and $ASInit(as)$. Therefore we can choose $(as, H, R) := (as_0, h_0, R_0)$.
- For *Correctness*, assume that both $BS((cs', H'), (as', H', R'))$ and $HCOp_{p,j}((cs, H), (cs', H'))$ hold. We have to find (as, H, R) with $BS((cs, H), (as, H, R))$ and $HAOp^*((as, H, R), (as', H', R'))$. Now for all concrete operations H is a prefix of H' : either $H' = H$ or $H' = H \cap \langle e \rangle$ for invoking and returning operations that add an event e . Prefix closedness of possibilities (Prop. 1) gives a reachable state (as, H, R) for the prefix H of H' with $HAOp^*((as, H, R), (as', H', R'))$. Again, if $H \neq \langle \rangle$, this state already satisfies BS . Otherwise, like for finalisation, we have to choose the initial state. \square

The theorem gives a backward simulation which matches an invoke operation $COp_{p,j}$ to a sequence $Lin^* ; Inv_{p,abs(j)} ; Lin^*$, where $Lin = \bigcup Lin_{p,i}$. Similarly, return operations match a sequence of linearisation steps with one return in the middle (since only such sequences add the right event to H). Other steps are matched to an empty sequence of abstract steps.

Theorem 3 specialises general completeness results, which imply that backward simulations and history variables are jointly complete for data refinement (these can be adapted to our formalism from [1], or more directly from [19], Theorem 5.6). However, all general completeness proofs add history variables, which record the full history of *all concrete states*. Theorem 3 shows, that for linearisability, the only history variable ever needed is the history needed to define linearisability (i.e. possibilities) itself.

5 Backward Simulation for the Case Study

The theory given in the last section ensures that any linearisable algorithm can be verified using a backward simulation BS . However, it does not tell us how to find such a

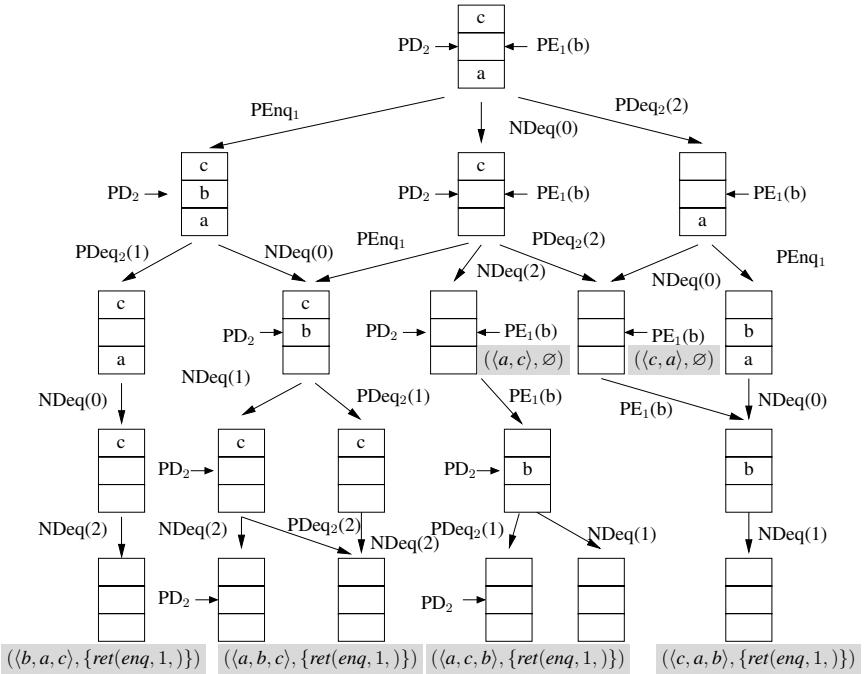


Fig. 1. Observation tree for an example state cs

relation between concrete states (cs, h) and abstract states (as, H, R) . As the abstract state in our case study consists of the queue variable q only, we also write (q, H, R) for the state of HA . As a first observation, the finalisation condition requires $h = H$ and thus we can split BS into the part relating state spaces and that of relating histories.

$$BS((cs, h), (q, H, R)) \hat{=} B(cs, q, R) \wedge h = H$$

The key insight we now need is that for finding backward simulations one has to analyse the *observations made by future behaviours*.

To explain the approach consider the example state (the history for this state was given at the start of Section 3) cs shown at the top of Figure 1. The state shows a situation where the array has been filled with two elements, $AR(0) = a$ and $AR(2) = c$. Furthermore process 1 is running an enqueue operation that tries to enqueue the element b at position 1, which has reached $pc(1) = E2$, but then has been preempted. We call such an operation with $pc(1) = E2$ a *pending enqueue*, and write $PE_1(b)$ in the figure to indicate it. Note that the “gap” in the array is due to this enqueue: it has increased the global *back* pointer before the enqueue of c , but has not executed statement $E2$ yet. In addition there is a *pending dequeue* of process 2 (PD_2) currently looking at position 1 as well. Such a dequeue operation has already initialised its *lback* ($pc \neq D1$), but has not yet successfully retrieved an element ($pc \neq D6$, and if $pc = D4$ then still $lv = \text{null}$).

To define B we now have to find out what possible abstract queue states this concrete state could correspond to. For this we look at observations made about this state when

proceeding with executions on it. The *observation tree* shows all future executions from this state when new *observers* are started. An observer gives us information about the elements in the data structure, most often by extracting data from it. For our queue, the observers are dequeue operations. Processes currently running (like the enqueue) might or might not be continued.

First, consider the leftmost branch. It describes the following steps: (1) the pending enqueue of process 1 runs to completion ($PEnq_1$), then (2) the pending dequeue runs to completion and returns the element in position 1 which is b ($PDeq_2(1)$), (3) a new dequeue is started (of whatever process), runs to completion and returns the element stored in position 0 which is a ($NDeq(0)$), and (4) another new dequeue starts, completes and returns the element in position 2 which is c . Hence from the point of view of these dequeues the queue contents has been $\langle b, a, c \rangle$. Note that we do not start any new enqueues, we just *observe* the existing state of the queue.

The rightmost branch executes pending operations in a different order (first the dequeue and then the enqueue) and again runs two observing dequeues. Here, we see that the queue is $\langle c, a, b \rangle$. Different future executions thus give different orderings of queue elements. Still, the order is not arbitrary: for instance $\langle b, c, a \rangle$ is impossible. We see that it is not only the current state of the array which determines the queue content, but also the pending enqueues and dequeues, and their current position into the array. Hence we cannot define B as a *function* from concrete to abstract state since this would contradict one or the other run. In summary, the backward simulation we look for must relate the current state cs to any queue that is possible in a future observation.

We still have to determine the R -components B relates concrete states to. Recall that R collects linearisation points. Again, general advice on finding a backward simulation is to *defer decisions as far as possible to the future* (this observation is not specific to linearisability or concurrency, see [3]). For our case, we delay any linearisation point that still can be executed to the future, i.e. we do *not* add it to set R . This is possible for pending dequeues. These can linearise at the time they swap the element: they have a *definite* linearisation point in the sense that we can attach it to line $D3$ when they swap a non-null element. However, enqueue operations cannot linearise in the future, since they would put the element in the wrong place in the queue. We find, that enqueue already *potentially* linearises when it executes $E1$, but only if the future run considered executes the operation to the end. In other runs, linearisation will happen when the element is actually inserted at line $E2$.

These considerations now help us towards defining B . We write $NDeq(n)(cs, cs')$ ³ to mean that a new (observer) dequeue is started, returns the element in array position n and brings the concrete state from cs to cs' . Similarly, we write $PDeq_p(n)(cs, cs')$ to say the same for an already running (pending) dequeue of process p , and finally $PEnq_p(cs, cs')$ for the completion of a pending enqueue. The actual definition of B *recursively* follows the paths of the tree and has to consider four cases:

- The array is empty. Then the queue is empty as well and the set R consists of return events for those processes which have definitely achieved their effect (denoted $outs(cs)$). In our case, these are all the enqueues after their store (at $E3$), and the dequeues after the non-*null* swap (at $D6$ or at $D4$, when $lv \neq null$).

³ The web presentation [17] gives a formal definition.

- An observing dequeue (newly started) returns the element in position n of the array. All elements below n must be *null*. The corresponding abstract queue thus has $AR[n]$ as its first element. The rest of the queue (and of B) is defined by recursion.
- A pending dequeue finishes and returns the element in position n of the array. Thus again one of the corresponding abstract queues has $AR[n]$ as first element. The rest of the queue (and B) is defined by recursion.
- A pending enqueue finishes and the corresponding return event is already in R . Then the effect on the abstract queue has already taken place, i.e., $ret(p, enq,) \in R$. B is defined by recursion using the same queue q , but removing the return event from R .

Putting into one definition (and taking as abstract state *as* the queue state q) we get

$$\begin{aligned} B(cs, q, R) := & (\forall i : \mathbf{N} \bullet AR[i] = null) \wedge q = \langle \rangle \wedge R = outs(cs)) \\ & \vee (\exists q', n \bullet q = \langle AR[n] \rangle \cap q' \wedge (NDeq(n) \circ B)(cs, q', R)) \\ & \vee (\exists q', p, n \bullet q = \langle AR(n) \rangle \cap q' \wedge (PDeqp(n) \circ B)(cs, q', R)) \\ & \vee (\exists p \bullet ret(enq, p,) \in R \wedge (PEnqp \circ B)(cs, q, R \setminus \{ret(enq, p,)\})) \end{aligned}$$

Applying this technique to our example state cs in the root of Figure 1 gives a total of six pairs (q, R) with $B(cs, q, R)$. These are written with shaded background at those nodes of the tree where the array is empty.

Note that the definition of B is well-founded: $PEnq$ removes a pending enqueue process (and adds one element to the array), $PDeq$ and $NDeq$ each remove an array element. The corresponding well-founded order $<_B$ plays a central role in the correctness proofs of the next section.

6 Verification with KIV

KIV [23] is an interactive verifier, based on structured algebraic specifications using higher-order logic (simply typed lambda-calculus). Crucial features of KIV used in the proofs here are the following.

- Proofs in KIV are explicit proof trees of sequent calculus which are graphically displayed and can be manipulated by pruning branches, or by replaying parts of proofs after changes. This is of invaluable help to analyse and efficiently recover from failed proof attempts due to incorrect theorems, which is typically the main effort when doing a case study like the one here.
- KIV implements correctness management: lemmas can be freely used before being proved. This allows to focus on difficult theorems first, which are subject to corrections. Changing a lemma invalidates those proofs only, that actually used it.
- KIV uses heuristics (e.g. for quantifier instantiation and induction) together with conditional higher-order rewrite rules to automate proofs. The rules are compiled into functional code, which runs very efficiently even for a large number of rules: the case study here uses around 2000 rules, 1500 of these were inherited from KIV's standard library of data types.

KIV was used to verify the completeness result for backward simulation as well as to prove the resulting proof obligations for the queue case study. A web presentation of all specifications and proofs can be found online [17]. The completeness proof follows the proof given in Section 4, the difficult part is Theorem 1.

The correctness of the queue implementation is proved by instantiating the backward simulation relation B with the concrete operations of the Herlihy-Wing queue which were sketched in Section 2. This results in proof obligations that are instances of the backward simulation as given in Definition 3.

The interesting proof obligations for the case study are the correctness conditions for each operation. These can be written as⁴

$$(HCOp_{p,j} \circ BS)((cs, H), (q', R', H')) \Rightarrow \\ \exists q, R \bullet BS((cs, H), (q, R, H)) \wedge HAOp^*((q, R, H), (q', R', H'))$$

A suitable sequence of abstract operations $HAOp^*$ that fixes q and R is easy to determine in most cases: for invoking and returning operations it is just the corresponding abstract invoke and return. For all other operations, except $enq1_p$, $enq2_p$ and $deq3t_p$ (the case of $deq3$, where the swap is with a non-null element), the sequence is empty. These correspond to cases where the observation tree for the current state cs is not changed by the operation. For $deq3t_p$ and $enq2_p$ the sequence is the linearisation step $Lin_{deq,p}$ resp. $Lin_{enq,p}$. These two operations reduce the observation tree to one of its branches. The only difficult case is when $COp_{p,j}$ is $enq1_p$ which is explained below. The choice of $HAOp^*$ simplifies the proof obligation to

$$(COp_{p,j} \circ B)(cs, q', R') \Rightarrow B(cs, q, R)$$

The simplicity of the changes to the observation tree is then reflected by the simplicity of the proofs: they all are proven by well-founded induction over $<_B$, followed by a case split over the definition of B . This gives a trivial base case and three recursive cases for each $PN \in \{PDeq_q(n), PEnq_q, NDeq(n)\}$. The resulting goals can be closed immediately with the induction hypothesis by noting that $COP_{p,j}$ and PN always commute. The only exception is $deq3t_p$ which needs an auxiliary lemma that $PEnq_q \circ deq3f_p$ commutes with every PN . This case crucially relies on the obvious invariant that there may be no more than one pending enqueue process for each array element.

The difficult case is $enq1_p$ which adds a new pending enqueue process, and has to deal with a potential linearisation point. To see what happens, consider the example shown in Fig. 2. It shows a situation on the left where an element a is in the array and process 2 is pending with element b , together with the possible observations (q, R) returned by the simulation B . Process 1 then executes $enq1_1(cs, cs')$ and becomes pending too with element c . This is shown on the right, together with the possible pairs (q', R') such that $B(cs', q', R')$.

The pairs (q, R) before the operation are exactly the subset of those pairs (q', R') where $ret(1, enq,) \notin R'$, i.e., the potential linearisation point has not been executed. For this case simulation is trivial, choosing the empty sequence as $HAOp^*$. The difficult cases have $ret(1, enq,) \in R'$. As the last result with $q' = \langle a, c, b \rangle$ shows, the element

⁴ For easier readability, we leave out the invariants of the two data types.

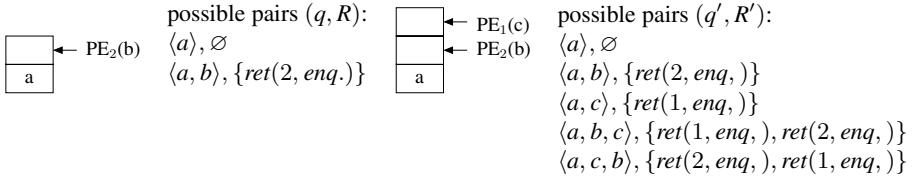


Fig. 2. Results of B before and after executing $enq1_2$

c may be observed to be *not* the last element of the queue. This demonstrates that one linearisation step with c is not sufficient on the abstract level. Instead the right choice for (q, R) is $(\langle a \rangle, \emptyset)$, and both linearisation steps $Lin_{1,enq} \circ Lin_{2,enq}$ are necessary as $HAOp^*$. This exploits the fact that the potential linearisation of process 2 *may not have been executed*, and can still be executed after the one for process 1.

In general, the element c enqueueued by some process p may be observed in any place behind the current elements of the array: we have $q' = q \cap \langle c \rangle \cap q_2$, where q_2 only consists of elements that pending enqueues will add in the future. Adding $\langle c \rangle \cap q_2$ corresponds to a sequence of abstract linearisation steps $LinE_{p,\underline{r}} := Lin_{p,enq}; Lin_{r_1,enq} \circ \dots \circ Lin_{r_n,enq}$. For the last result of the example, $\underline{r} = \langle 2 \rangle$ and $q_2 = \langle a \rangle$. Therefore we strengthen the proof obligation for $enq1_p$ to

$$(enq1_p \circ B)(cs, q', R') \wedge ret(p, enq,) \in R' \Rightarrow \\ \exists q, q_2, R, \underline{r} \bullet B(cs, q, R) \wedge LinE_{p,\underline{r}}((q, R, H), (q', R', H))$$

Again the proof follows the standard well-founded induction scheme over $<_B$. The difficult case occurs when unfolding B executes $PEnq_p$ for the same process p . This case requires another induction to prove that $enq1_p \circ PEnq_p$ commutes with all PN . This works except for a new dequeue process that removes the element just added by process p , which can only happen for an empty array. We finally complete the proof of $enq1_p$ by showing that the observable queues for an empty array consist of some (or none) of the elements of pending enqueues (in any order).

7 Related Work

Our work gives a general and practically applicable method for proving linearisability. It should be contrasted with other methods of proving linearisability which fall into several classes.

First, there is work on model checking linearisability, e.g. [5] for checking a specific algorithm or [4] for a general strategy. These approaches are very good at finding counter examples when linearisability is violated. However, these methods only check short sequences of (usually two or three) operations by exploring all possibilities of linearisation points, so they do not give a full proof. They also do not yield any explanation of why a certain implementation is indeed linearisable.

Work on full proofs has analysed several classes of increasing complexity, where figuring out simulations (in particular thread-local ones, that exploit the symmetry of all processes executing the same operations) becomes increasingly difficult.

The simplest standard class of algorithms has an abstraction *function*, and all linearisation points can be fixed to be specific instructions of the code of an algorithm (often atomic compare-and-swap (CAS) instructions are candidates). A variety of approaches for such algorithms have been developed: [12] uses IO-Automata refinement and interactive proofs with PVS, [27] executes abstract operations as “ghost-code” at the linearisation point, arguing informally that linearisability is implied. Proof obligations for linearisability have also been verified using shape analysis [2].

Our own work in [7] gave step-local forward simulation conditions for this standard case. Conditions were optimised for the case where reasoning about any number of processes can be reduced to thread-local reasoning about one process and its environment abstracted to one other process. It mechanised proofs that these are indeed sufficient to prove linearisability.

A second, slightly more difficult class are algorithms where the linearisation point is non-deterministically one of several instructions, the Michael-Scott queue ([20]) being a typical example. [9] has given a solution using backward and forward simulation, Vafeiadis [27] uses a prophecy variable as additional ghost code. Our work here shows that backward simulation alone is sufficient.

A third, even more difficult class are algorithms that use observer operations that do not modify the abstract data structure. Such algorithms often have no definite linearisation point in the code. Instead steps of *other* processes linearise. The standard example for this class is Heller et al’s “lazy” implementation of sets [13]. There, the *contains* algorithm that checks for membership in the set has no definitive linearisation point. Based on the idea that linearisation of such operations can happen at any time during its execution, [28] develops the currently most advanced automated proof strategy for linearisability in the Cave tool.

Our work in [8] gives thread-local, step-local conditions for this class, and verifies Heller et al’s lazy set. Mechanised proofs that these conditions can be derived from the general theory given here are available on the Web too [16].

All these three classes, where mechanised proofs have been attempted, had an abstraction function, so different possibilities for one concrete state could only differ in the possible linearisation points that have been executed (our set R of return events). However, the Herlihy-Wing queue is just the simplest example that falls outside of these classes. We have chosen it here since it is easy to explain, not because it is practically relevant. One of the practically relevant examples is the elimination queue [21], which to our knowledge is currently the most efficient lock-free queue implementation. This example has some striking similarities to the case study considered here. Verifying this case study is future work, however it seems clear that it can be verified using exactly the same proof strategy as shown here.

For this most complex class only pencil-and-paper approaches exist to proving linearisability, so our proof of the Herlihy-Wing queue is the first that mechanises such a proof (and even a full proof, not just the verification of proof obligations justified on paper) for this algorithm. Our proof is step-local in considering stepwise simulation. Even for simpler classes many proof approaches so far have resorted to global arguments about the past, either informally e.g. [13], [20], [29], using explicit traces [22] or with temporal past operators [11].

Herlihy and Wing's own proof in [24] also uses such global arguments: first, it adds a global, auxiliary variable to the code. The abstraction relation based on this variable is not a simulation. Therefore they have to use global, queue-specific lemmas (Lemmas 11 and 12) about normalised derivations to ensure that it is possible to switch from one (q, R) to another (q', R') in the middle of the proof.

8 Conclusion and Future Work

In this paper, we have presented a sound and complete proof technique for linearisability of concurrent data structures. We have exemplified our technique on the Herlihy and Wing queue which is one of the most complex examples of a linearisable algorithm. Except for pen-and-paper proofs no-one has treated this example before, in particular none of the partially or fully automatic approaches to proving linearisability. Both the linearisability proof for the queue and the general soundness and completeness proof for our technique have been mechanised within an interactive prover.

The proof strategy given here is complete, but still not optimal in terms of reduction of proof effort: in particular, we have to encode the algorithms as operations, and just like in Owicky-Gries style proofs we require specific assertions for every particular value of the program counter. Rely-Guarantee reasoning [15] can help to reduce the number of necessary assertions and we have already developed an alternative approach based on Temporal Logic that used Relys and Guarantees. That approach can currently handle the standard class of algorithms for linearisability, though it has advantages for proving the liveness property of lock-freedom [24] and has been used to verify the hard case-study of Hazard pointers [25]. Integrating both approaches remains future work.

Our approach is also not fully optimal for heap-based algorithms, where the use of concurrent versions of separation logic (e.g. RGsep [28] or HLRG [11]) helps to avoid disjointness predicates between (private) portions of the heap, and gives heap-local reasoning.

Finally, there is a recent trend to generalise linearisability to general refinement of concurrent objects [10], [26], where the abstract level is not required to execute one abstract operation. We have not yet studied these theoretically interesting generalisations, since they are not needed for our examples. This – as well as techniques for optimising our approach with respect to proof effort – is left for future work.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. *Theoretical Computer Science* 2, 253–284 (1991)
2. Amit, D., Rinotzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison Under Abstraction for Verifying Linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
3. Banach, R., Schellhorn, G.: Atomic Actions, and their Refinements to Isolated Protocols. *FAC* 22(1), 33–61 (2010)
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proceedings of PLDI, pp. 330–340. ACM (2010)

5. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model Checking of Linearizability of Concurrent List Implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010)
6. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Springer (May 2001)
7. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically verified proof obligations for linearizability. ACM Trans. Program. Lang. Syst. 33(1), 4 (2011)
8. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying Linearisability with Potential Linearisation Points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)
9. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal Verification of a Practical Lock-Free Queue Algorithm. In: de Frutos-Escríg, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
10. Filipovic, I., O’Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theoretical Computer Science 411(51–52), 4379–4398 (2010)
11. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about Optimistic Concurrency Using a Program Logic for History. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 388–402. Springer, Heidelberg (2010)
12. Groves, L., Colvin, R.: Derivation of a scalable lock-free stack algorithm. ENTCS 187, 55–74 (2007)
13. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A Lazy Concurrent List-Based Set Algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
14. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
15. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP 1983, pp. 321–332. North-Holland (1983)
16. Web presentation of linearizability theory and the lazy set algorithm (2010),
<http://www.informatik.uni-augsburg.de/swt/projects/possibilities.html>
17. Web presentation of KIV proofs for this paper (2011),
<http://www.informatik.uni-augsburg.de/swt/projects/Herlihy-Wing-queue.html>
18. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
19. Lynch, N., Vaandrager, F.: Forward and Backward Simulations – Part I: Untimed systems. Information and Computation 121(2), 214–233 (1995)
20. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on Principles of Distributed Computing, pp. 267–275 (1996)
21. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: SPAA, pp. 253–262. ACM (2005)
22. O’Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pp. 85–94 (2010)
23. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Automated Deduction—A Basis for Applications, Interactive Theorem Proving, vol. II, ch. 1, pp. 13–39. Kluwer (1998)
24. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal Logic Verification of Lock-Freedom. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 377–396. Springer, Heidelberg (2010)

25. Tofan, B., Schellhorn, G., Reif, W.: Formal Verification of a Lock-Free Stack with Hazard Pointers. In: Cerone, A., Pihlajaasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 239–255. Springer, Heidelberg (2011)
26. Turon, A., Wand, M.: A separation logic for refining concurrent objects. In: POPL, vol. 46, pp. 247–258. ACM (2011)
27. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
28. Vafeiadis, V.: Automatically Proving Linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
29. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP 2006, pp. 129–136. ACM (2006)

Synchronisation- and Reversal-Bounded Analysis of Multithreaded Programs with Counters

Matthew Hague^{1,2} and Anthony Widjaja Lin²

¹ LIGM (Université Paris-Est), LIAFA (Université Paris Diderot) & CNRS

² Oxford University, Department of Computer Science

Abstract. We study a class of concurrent pushdown systems communicating by both global synchronisations and reversal-bounded counters, providing a natural model for multithreaded programs with procedure calls and numeric data types. We show that the synchronisation-bounded reachability problem can be efficiently reduced to the satisfaction of an existential Presburger formula. Hence, the problem is NP-complete and can be tackled with efficient SMT solvers such as Z3. In addition, we present optimisations to make our reduction practical, e.g., heuristics for removing or merging transitions in our models. We provide optimised algorithms and a prototypical implementation of our results and perform preliminary experiments on examples derived from real-world problems.

1 Introduction

Pushdown systems (PDS) are a popular abstraction of sequential programs with recursive procedure calls. Verification problems for these models have been extensively studied (e.g. [7, 17]) and they have been successfully used in the model checking of sequential software (e.g. [3, 5, 37]).

However, given the ubiquity and growing importance of concurrent software (e.g. in web-servers, operating systems and multi-core machines), coupled with the inherent non-determinism and difficulties in anticipating all concurrent interactions, the verification of concurrent programs is a pressing problem. In the case of concurrent pushdown systems, verification problems quickly become undecidable [33]. Because of this, much research has attempted to address the undecidability, proposing many different approximations, and restrictions on topology and communication behaviour (e.g. [29, 8–10, 35, 34, 21, 25]). A technique that has proved popular in the literature is that of *bounded context-switches* [34].

Bounded context-switching uses the observation that many real-world bugs require only a small number of inter-thread communications. It is known that, if the number of communications is bounded to a fixed k , reachability checking of pushdown systems becomes NP-complete [26]. The utility of this approach has been demonstrated by several successful implementations (e.g. [4, 30, 36]).

In addition to recursive procedure calls, numeric data types are an important feature of programs. By adding counters to pushdown systems one can accurately model integer variables and, furthermore, abstract certain data structures – such as lists – by tracking their size. It is well known that finite-state machines

augmented even with only two counters leads to undecidability of the simplest verification problems. One way to retain decidability of reachability is to impose an upper bound r on the number of reversals between incrementing and decrementing modes for each counter (cf. [12, 23]).

This restriction can be viewed in at least two ways (cf. [12, 24]). First, in the spirit of bounded-context switches, it provides a generalisation of bounded model checking – a successful verification technique which exploits the fact that many bugs occurring in practice are “shallow” (cf. [14]). Secondly, many counting properties — such as checking the existence of a computation where the number of calls to the functions f_1 , f_2 , f_3 , and f_4 are the same — require no reversals (e.g. the number of memory allocations equals the number of frees). Similar counting properties (and their model checking problems) have been studied in many other contexts (cf. [27] and references therein).

In this paper, we study the problem of verifying reachability over a program model incorporating concurrency, numeric data types, and recursions. Our contributions are as follows:

1. We propose a concurrent extension of pushdown systems with reversal-bounded counters that communicate through shared counters and global synchronisations, and prove that the notion of global synchronisations subsumes context-bounded model checking.
2. We show that reachability checking for these systems is in NP, by reduction to existential Presburger, handled by efficient SMT solvers such as Z3 [13].
3. We provide several new optimisation techniques, including a minimisation routine for pushdown systems, that are crucial in making our reductions feasible in practice. These techniques keep the size of the computation objects small throughout reduction, while also producing smaller output formulas.
4. Finally, we provide two optimised, prototypical tools using these techniques. The first translates a simple programming language into our model, while the second performs our reduction to existential Presburger. We demonstrate the efficacy of our tools on several real-world problems.

The full version of this paper and tool implementations and benchmarks can be obtained from the authors’ homepages.

Related Work. In recent work [20], we showed that reachability analysis for pushdown systems with reversal-bounded counters is NP-complete. We provided a prototypical implementation of our algorithm and obtained encouraging results on examples derived from Linux device drivers.

Over reversal-bounded counter systems (without stack), reachability is NP-complete but becomes NEXP-complete when the number of reversals is given in binary [25]. On the other hand, when the numbers of reversals and counters are fixed, the problem is solvable in P [19]. The techniques developed by [19, 22], which reason about the maximal counter values, are very different to our techniques, which exploit the connection to Parikh images of pushdown automata (first explicated in Ibarra’s original paper [23] though not in a way that gives optimal complexity or a practical algorithm).

Context-bounded model checking was introduced in 2005 by Qadeer and Rehof [34, §8, 32]. It has then been used in many different settings and many different generalisations have been proposed. For example, one may consider phase-bounds [38], ordered multi-stack machines [1], bounded languages [18], dynamic thread creation [2] and more general approaches [28].

In recent, independent work, Esparza *et al.* used a reduction to existential Presburger to tackle a generalisation of context-bounded reachability checking for multithreaded programs without counters [15]. Their work, however, does not allow the use of counters and it is not clear whether our global synchronisation conditions can be simulated succinctly in their framework.

Organisation. In §2 we define the models that we study. We prove decidability of the synchronisation-bounded reachability problem in §3. In §4 we show that synchronisation-bounded model checking subsumes context-bounded model checking. Our optimisations are presented in §5. In §6 we describe our implementation and experimental results. Finally, we conclude in §7.

2 Model Definition

In this section, we define the models that we study. For a vector $\mathbf{v} = (v_1, \dots, v_n)$, we write $\mathbf{v}(i)$ to access v_i . For a formula θ over variables (x_1, \dots, x_n) we write $\theta(v_1, \dots, v_n)$ to substitute the values v_1, \dots, v_n for the variables x_1, \dots, x_n respectively. Given an alphabet $\Gamma = \{\gamma_1, \dots, \gamma_m\}$ and a word $w \in \Gamma^*$, we write $\mathbb{P}(w)$ to denote a tuple with $|\Gamma|$ entries where the i th entry counts the number of occurrences of γ_i in w . Given a language $\mathcal{L} \subseteq \Gamma^*$, we write $\mathbb{P}(\mathcal{L})$ to denote the set $\{ \mathbb{P}(w) \mid w \in \mathcal{L} \}$. We say that $\mathbb{P}(\mathcal{L})$ is the *Parikh image* of \mathcal{L} .

Pushdown Automata. A *pushdown automaton* \mathcal{P} is a tuple $(\mathcal{Q}, \Sigma, \Gamma, \Delta, q_0, \mathcal{F})$ where \mathcal{Q} is a finite set of control states, Σ is a finite stack alphabet with a special bottom-of-stack symbol \perp , Γ is a finite output alphabet, $q_0 \in \mathcal{Q}$ is an initial state, $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states, and $\Delta \subseteq (\mathcal{Q} \times \Sigma) \times \Gamma^* \times (\mathcal{Q} \times \Sigma^*)$ is a finite set of transition rules. We will denote a transition rule $((q, a), \gamma, (q', w'))$ using the notation $(q, a) \xrightarrow{\gamma} (q', w')$. Note that $\gamma \in \Gamma^*$ is a sequence of output characters. This is for convenience, and optimisation. We can reduce this to single output characters using intermediate control states or stack characters. Note, a *pushdown system* is a pushdown automaton without a set of final states.

A configuration of \mathcal{P} is a tuple (q, w) , where $q \in \mathcal{Q}$ and $w \in \Sigma^*$ are the control state and stack contents. We say that a configuration (q, aw) has a *head* q, a . There exists a transition $(q, aw) \xrightarrow{\gamma} (q', w'w)$ of \mathcal{P} whenever $(q, a) \xrightarrow{\gamma} (q', w') \in \Delta$. We call a sequence $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \dots \xrightarrow{\gamma_m} c_m$ a *run* of \mathcal{P} . It is accepting if $c_0 = (q_0, \perp)$ and $c_m = (q, w)$ with $q \in \mathcal{F}$. Let $\mathcal{L}(\mathcal{P})$ be the set of words labelling accepting runs. Finally, we write $c \rightarrow^* c'$ if there is a run from c to c' .

Pushdown Systems with Counters. A pushdown system with counters is a pushdown system which, in addition to the control states and the stack, has a number of counter variables. These counters may be incremented, decremented and compared against constants (given in binary).

An *atomic counter constraint* on counter variable $X = \{x_1, \dots, x_n\}$ is an expression of the form $x_i \sim c$, where $c \in \mathbb{Z}$ and $\sim \in \{\langle, \rangle, =\}$. A *counter constraint* $\theta(x_1, \dots, x_n)$ on X is a boolean combination of atomic counter constraints on X . Let $Const_X$ denote the set of counter constraints on X .

A *pushdown system with n counters* (n -PDS) \mathcal{P} is a tuple $(\mathcal{Q}, \Sigma, \Gamma, \Delta, X)$ where \mathcal{Q} is a finite set of control states, Σ is a finite stack alphabet, Γ is a finite output alphabet, $X = \{x_1, \dots, x_n\}$ is a set of n counter variables, and $\Delta \subseteq (\mathcal{Q} \times \Sigma \times Const_X) \times \Gamma^* \times (\mathcal{Q} \times \Sigma^* \times \mathbb{Z}^n)$ is a finite set of transition rules. We will denote a rule $((q, a, \theta), \gamma, (q', w', \mathbf{u}))$ using $(q, a, \theta) \xrightarrow{\gamma} (q', w', \mathbf{u})$.

A configuration of \mathcal{P} is a tuple (q, w, \mathbf{v}) , where $q \in \mathcal{Q}$ is the current control state, $w \in \Sigma^*$ is the current stack contents, and $\mathbf{v} = (v_1, \dots, v_n) \in \mathbb{N}^n$ gives the current valuation of the counter variables x_1, \dots, x_n respectively. There exists a transition $(q, aw, \mathbf{v}) \xrightarrow{\gamma} (q', w'w, \mathbf{v}')$ of \mathcal{P} whenever

1. $(q, a, \theta) \xrightarrow{\gamma} (q', w', \mathbf{u}) \in \Delta$, and
2. $\theta(v(1), \dots, v(n))$ is true, and
3. $v'(i) = v(i) + \mathbf{u}(i) \geq 0$ for all $1 \leq i \leq n$.

Communicating Pushdown Systems with Counters. Given $\mathcal{Q}_1, \dots, \mathcal{Q}_m$, let $Y = \{y_1, \dots, y_m, y'_1, \dots, y'_m\}$ be a set of control state variables such that, for each i , y_i, y'_i range over \mathcal{Q}_i . Then, an *atomic state constraint* is of the form $y_i = q$ for some $y_i \in Y$ and $q \in \mathcal{Q}_i$. A *synchronisation constraint*, written $\delta(y_1, \dots, y_m, y'_1, \dots, y'_m)$, is a boolean combination of atomic state constraints. For example, let $n = 3$ and consider the constraint

$$(y_1 = q_1 \wedge (y'_1 = q_1 \wedge y'_2 = q_2 \wedge y'_3 = q_3)) \vee \\ (y_1 = r_1 \wedge (y'_1 = r_1 \wedge y'_2 = r_2 \wedge y'_3 = r_3)) .$$

This allows synchronisations where, whenever the first process has control state q_1 , the other processes can simultaneously move to q_i (for all $1 \leq i \leq 3$), whereas, if process one has control state r_1 , the processes move to states r_i instead. Let $StateCons_{\mathcal{Q}_1, \dots, \mathcal{Q}_m}$ be the set of synchronisation constraints for $\mathcal{Q}_1, \dots, \mathcal{Q}_m$.

Definition 1 (n-SyncPDSR). Given a finite output alphabet Γ and set of n counter variables X , a system of communicating pushdown systems with n counters \mathbb{C} is a tuple $(\mathcal{P}_1, \dots, \mathcal{P}_m, \Delta_g, X, r)$ where, for all $1 \leq i \leq m$, \mathcal{P}_i is a pushdown system $(\mathcal{Q}_i, \Sigma_i, \Gamma, \Delta_i, X)$ with n counters, and $\Delta_g \subseteq StateCons_{\mathcal{Q}_1, \dots, \mathcal{Q}_m} \times Const_X \times \mathbb{Z}^n$ is a finite set of synchronisation constraints, and $r \in \mathbb{N}$ is a natural number given in unary.

Notice that a system of communicating pushdown systems share a set of counters. A configuration of such a system is a tuple $(q_1, w_1, \dots, q_m, w_m, \mathbf{v})$ where each (q_i, w_i, \mathbf{v}) is a configuration of \mathcal{P}_i . We have $(q_1, w_1, \dots, q_m, w_m, \mathbf{v}) \xrightarrow{\gamma} (q'_1, w'_1, \dots, q'_m, w'_m, \mathbf{v}')$ whenever,

1. for some $1 \leq i \leq m$, we have $(q_i, w_i, \mathbf{v}) \xrightarrow{\gamma} (q'_i, w'_i, \mathbf{v}')$ is a transition of \mathcal{P}_i and $q_j = q'_j$ and $w_j = w'_j$ for all $j \neq i$, or

2. $\gamma = \varepsilon$ and $w_i = w'_i$ for all $1 \leq i \leq m$ and $(\delta, \theta, \mathbf{u}) \in \Delta_g$ with
 - (a) $\delta(q_1, \dots, q_m, q'_1, \dots, q'_m)$ is true, and
 - (b) $\theta(\mathbf{v}(1), \dots, \mathbf{v}(n))$ is true, and
 - (c) $\mathbf{v}'(i) = \mathbf{v}(i) + \mathbf{u}(i) \geq 0$ for all $1 \leq i \leq n$.

We refer to these two types of transition as *internal* and *synchronising* respectively. A run of \mathbb{C} is a run $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \dots \xrightarrow{\gamma_m} c_m$.

Bounding Runs. During a run, the counter is in a non-decrementing mode if the last value-changing operation on that counter was an increment. Similarly, a counter may be in a non-incrementing mode. The number of reversals of a counter during a run is the number of times the counter changes from an incrementing to a decrementing mode, and vice versa. For example, if the values of a counter x in a path are $1, 1, 1, 2, 3, 4, 4, \overline{4}, \overline{3}, 2, \overline{2}, \overline{3}$, then the number of reversals of x is 2 (reversals occur in between the overlined positions). This sequence has three *phases* (i.e. subpaths interleaved by consecutive reversals or end points): non-decrementing, non-incrementing, and finally non-decrementing.

Definition 2 (r -Reversal-Bounded). A run $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \dots \xrightarrow{\gamma_m} c_m$ is r -reversal-bounded whenever we can partition $c_0 c_1 \dots c_m$ into $C_1 \dots C_r$ such that for all $1 \leq p \leq r$, there is some $\sim \in \{\leq, \geq\}$ such that for all $c_j c_{j+1}$ appearing together in C_p , we have $c_j = (\dots, \mathbf{v}_j)$, $c_{j+1} = (\dots, \mathbf{v}_{j+1})$, and for all $1 \leq i \leq n$, $\mathbf{v}_j(i) \sim \mathbf{v}_{j+1}(i)$.

Finally, we define the notion of *synchronisation-bounded*. We show in Section 4 that this notion subsumes context-bounded model checking.

Definition 3 (k -Synchronisation-Bounded). A run π is k -synchronisation-bounded whenever π uses k or fewer synchronising transitions.

3 Synchronisation-Bounded Reachability

The *r -reversal and k -synchronisation-bounded reachability problem* for a given \mathbb{C} , bound r and k asks, for given configurations c and c' of \mathbb{C} , is there a k -synchronisation-bounded run of \mathbb{C} from c to c' using up to r reversals. We prove:

Theorem 1. For two bounds r and k given in unary, the r -reversal and k -synchronisation-bounded reachability problem for n -SyncPDS is NP-complete.

The proof extends the analogous theorem for r -reversal-bounded n-PDS [20]. We will construct, for each \mathcal{P}_i in \mathbb{C} , an over-approximating pushdown automaton \mathcal{P}'_i and use Verma *et al.* [40]¹ to obtain an existential Presburger formula Image_i giving the Parikh image of \mathcal{P}'_i . Finally, we add additional constraints such that a solution exists iff the reachability problem has a positive answer.

¹ It is well known that [40] contains a small bug, fixed by Barner [6]. See the full version for more details.

The encoding presented here is one of two encodings that we developed. This encoding is both simpler to explain and seems to be handled better by Z3 for almost all of our examples than the second encoding. However, the second encoding results in a smaller formula. Hence, we include both reductions as contributions, and present the second reduction in the full version of the paper.

The key difference between the encodings is where we store the number of synchronisations performed so far. In the first encoding, we keep a component g in each control state; thus, from each \mathcal{P} we build \mathcal{P}' with $|\mathcal{Q}| \times N_{\max} \times (k+1)$ control states, where \mathcal{Q} is set of control states of \mathcal{P} and N_{\max} is the number of mode vectors (where modes are defined below).

In the alternative encoding we put the number of synchronisations in the modes, resulting in $|\mathcal{Q}| \times (N_{\max} + k + 1)$ control states. This is important since our reduction is quadratic in the number of controls. Hence, if $k = 2$, the alternative results in pushdown automata a third of the size of the encoding presented here. However, the resulting formulas seem experimentally more difficult to solve.

Let $c = (q_1^0, w_1, \dots, q_m^0, w_m, \mathbf{v}_0)$ and $c' = (f_1, w'_1, \dots, f_m, w'_m, \mathbf{v}_f)$. By hard-coding the initial and final stack contents, we can assume that all $w_i = w'_i = \perp$.

Unfortunately, we cannot use the reduction for r -reversal-bounded n-PDS as a completely black box; hence, we will recall the relevant details and highlight the new techniques required. We refer the reader to the article [20] for further details. The correctness of the reduction is given in the full version of the paper.

The final formula `HasRun` will take the shape

$$\exists \mathbf{m}_1, \dots, \mathbf{m}_{N_{\max}} \exists z_1 \dots z_m \left(\begin{array}{l} \text{Init}(\mathbf{m}_1) \wedge \text{GoodSeq}(\mathbf{m}_1, \dots, \mathbf{m}_{N_{\max}}) \\ \quad \wedge \bigwedge_{1 \leq i \leq m} \text{Image}_i(z_i) \\ \quad \wedge \text{Respect}\left(\sum_{1 \leq i \leq m} z_i, \mathbf{m}_1, \dots, \mathbf{m}_{N_{\max}}\right) \\ \quad \wedge \text{OneChange}\left(\sum_{1 \leq i \leq m} z_i\right) \\ \quad \wedge \text{EndVal}\left(\sum_{1 \leq i \leq m} z_i\right) \wedge \text{Syncs}\left(\sum_{1 \leq i \leq m} z_i\right) \end{array} \right)$$

where the formulas $\text{OneChange}\left(\sum_{1 \leq i \leq m} z_i\right)$ and $\text{Syncs}\left(\sum_{1 \leq i \leq m} z_i\right)$ are the main differences with the single thread case. In addition, further adaptations need to be made within other aspects of the formula. We remark at this point that the user may add to `HasRun` an additional constraint on the Parikh images of runs — such as restricting to runs where the number of characters γ output is greater than the number of γ' .

The Mode Vectors. We begin with the vectors $\mathbf{m}_1, \dots, \mathbf{m}_{N_{\max}}$, which are unchanged from the case of r -reversal-bounded n-PDS. Let $d_1 < \dots < d_h$ denote all the numeric constants appearing in an atomic counter constraint as a part of the constraints in the \mathcal{P}_i . Without loss of generality, we assume that $d_1 = 0$ for convenience. Let $\text{REG} = \{\varphi_1, \dots, \varphi_h, \psi_1, \dots, \psi_h\}$ be a set of formulas defined as follows. Note that these formulas partition \mathbb{N} into $2h$ pairwise disjoint regions.

$$\varphi_i(x) \equiv x = d_i, \quad \psi_i(x) \equiv d_i < x < d_{i+1} \quad (1 \leq i < h), \quad \psi_h(x) \equiv d_h < x .$$

We call a vector in $\text{REG}^n \times [0, r]^n \times \{\uparrow, \downarrow\}^n$ a *mode vector*. Given a path π from configurations c to c' , we may associate a mode vector to each configuration in π . This vector records for each counter, which region its value is in, how many reversals it's used, and whether its phase is non-decrementing (\uparrow) or non-incrementing (\downarrow). Consider a sequence of mode vectors. A crucial observation is, once a change occurs to the mode information of a counter, the same information will not recur for that counter. For example, returning to the same region will incur an increase in the number of reversals. Thus, there are at most $N_{\max} := |\text{REG}| \times (r + 1) \times n = 2hn(r + 1)$ distinct mode vectors in any sequence.

Constructing \mathcal{P}'_i . We define the pushdown automata

$$\mathcal{P}'_i = (\mathcal{Q}'_i, \Sigma_i, \Gamma', \Delta'_i, (q_i^0, 1, 1), \{f_i\} \times [1, N_{\max}] \times [1, k + 1])$$

for each \mathcal{P}_i in \mathbb{C} . Note that each \mathcal{P}'_i has the same output alphabet Γ' . We assume that all \mathcal{Q}_i are pairwise disjoint. There are two main aspects to each \mathcal{P}'_i . First, we remove the counters. To replace them, we have \mathcal{P}'_i output any counter changes or tests that would have been performed. E.g. where \mathcal{P}_i would increment a counter, \mathcal{P}'_i will output a symbol $(\text{ctr}_j, 1, \dots)$ indicating (amongst other things) that counter ctr_j should be increased by 1. Furthermore, \mathcal{P}'_i guesses when, and keeps track of when, mode changes would have occurred. Secondly, we allow \mathcal{P}'_i to non-deterministically make synchronisations (instead of communicating, the effect of external threads is guessed). In this case, the control state change, along with the number of synchronisations performed thus far, will be output. In this way, \mathcal{P}'_i makes “visible” the counter tests, counter updates and synchronisations that would have been performed by \mathcal{P}_i on the same run. Constraints described later in `HasRun` ensure these operations are valid.

More formally, let $\mathcal{Q}'_i = \mathcal{Q}_i \times [1, N_{\max}] \times [1, k + 1]$ (that is, we add to \mathcal{Q}_i the current mode and synchronisation number). We define Γ' implicitly from the transition relation. In fact, Γ' is a (finite) subset of

$$\begin{aligned} \Gamma \cup \{ & (\text{ctr}_j, u, e, l) \mid j \in [1, n], u \in \mathbb{Z}, e \in [1, N_{\max}], l \in \{0, 1\} \} \\ & \cup (\text{Const}_X \times [1, N_{\max}]) \\ \cup & \bigcup_{1 \leq i \leq m} (\text{StateCons}_{\mathcal{Q}_1, \dots, \mathcal{Q}_m} \times \mathcal{Q}_i \times \mathcal{Q}_i \times [1, N_{\max}] \times [1, k + 1] \times \{0, 1\}). \end{aligned}$$

Characters (ctr_j, u, e, l) mean to add u to ctr_j , in mode e , where l indicates whether the counter action changes the mode vector. Characters (θ, e) indicate a counter test in mode e . Finally, characters (δ, q, q', e, g, l) indicate a use of synchronisation rule δ , changing \mathcal{P}_i from control state q to q' , in mode e with g synchronisations performed so far.

We define Δ'_i to be the smallest set such that, if $(q, a, \theta) \xrightarrow{\gamma} (q', w, \mathbf{u}) \in \Delta_i$ where $\mathbf{u} = (u_1, \dots, u_n)$ then for each $e \in [1, N_{\max}]$ and $g \in [1, k + 1]$, Δ'_i contains

$$((q, e, g), a) \xrightarrow{\gamma(\theta, e)(\text{ctr}_1, u_1, e, l) \dots (\text{ctr}_n, u_n, e, l)} ((q', e + l, g), w)$$

for all $l \in \{0, 1\}$ if $e \in [1, N_{\max}]$ and $l = 0$ otherwise. Thus, $l = 1$ signifies a mode changing transition.

These rules are the rules required in the single thread case. We need additional rules to reflect the multi-threaded environment. In particular, an external thread may change the mode, or a synchronising transition may occur. To account for this Δ'_i also has for each $q \in \mathcal{Q}_i$, $a \in \Sigma_i$, $e \in [1, N_{\max}]$, and $g \in [1, k+1]$,

$$((q, e, g), a) \xrightarrow{\varepsilon} ((q, e+1, g), a) \quad (*)$$

and, to model synchronisations, we have for all $q, q' \in \mathcal{Q}_i$, $e \in [1, N_{\max}]$, $g \in [1, k+1]$ and $(\delta, \theta, \mathbf{u}) \in \Delta_g$, when $i > 1$,

$$((q, e, g), a) \xrightarrow{(\delta, q, q', e, g, l)} ((q', e+l, g+1), a)$$

and when $i = 1$ and $\mathbf{u} = (u_1, \dots, u_n)$,

$$((q, e, g), a) \xleftarrow{(\delta, q, q', e, g, l)(\theta, e)(\mathbf{ctr}_1, u_1, e, l) \dots (\mathbf{ctr}_n, u_n, e, l)} ((q', e+l, g+1), a)$$

for all $l \in \{0, 1\}$ when $e \in [1, N_{\max}]$ and $l = 0$ otherwise. That is, \mathcal{P}'_i guesses the effect of non-internal transitions and \mathcal{P}'_1 is responsible for performing the required counter updates. Note that the information in the output character (δ, q, q', e, g, l) allows us to check that synchronising transitions take place in the same order and in the same modes across all threads.

Constructing The Formula. Fix an ordering $\gamma_1 < \dots < \gamma_l$ on Γ' . By f we denote a function mapping γ_i to i for each $i \in [1, l]$. Let \mathbf{z} denote a vector of l variables. The formula is `HasRun` given above, where `Init`, `GoodSeq`, `Respect`, and `EndVal` are defined as in the single thread case (using only variables which are unchanged from [20]); therefore, we describe them informally here, referring the reader to the full version of the paper for the full definitions. We convert each \mathcal{P}'_i to a context-free grammar (of cubic size) and use [40] to obtain `Image` $_i$ such that for each $\mathbf{n} \in \mathbb{N}^l$ we have $\mathbf{n} \in \mathbb{P}(\mathcal{L}(\mathcal{P}'_i))$ iff `Image` $_i(\mathbf{n})$ holds. Informally,

- `Init` ensures the initial mode vector \mathbf{m}_1 respects the initial configuration c ;
- `GoodSeq` ensures that the sequence of mode vectors $\mathbf{m}_1, \dots, \mathbf{m}_{N_{\max}}$ is valid. For example, if the direction of a counter changes, then an extra reversal is incurred on that counter;
- `Respect` requires that the counter tests and actions fired within a mode are allowed. For example, a subtraction may not occur on a counter in a non-decreasing phase, only one mode change action may occur per mode, and that counter tests only occur in sympathetic regions; and
- `EndVal` checks that the counter operations applied during the run leave each counter in the correct value, as given in the final configuration c' .

It remains for us to define `OneChange` and `Syncs`. We use `OneChange` to assert that only one thread may be responsible for firing the transition that changes a given mode of the counters to the next. That is,

$$\text{OneChange}(\mathbf{z}) \equiv \bigwedge_{\substack{(\mathbf{ctr}_j, u, e, 1) \\ (\mathbf{ctr}_j, u', e, 1) \\ u' \neq u}} z_{f(\mathbf{ctr}_j, u, e, 1)} > 0 \Rightarrow \left(\begin{array}{l} z_{f(\mathbf{ctr}_j, u, e, 1)} = 1 \\ \wedge z_{f(\mathbf{ctr}_j, u', e, 1)} = 0 \end{array} \right).$$

The role of Syncs is to ensure that the synchronising transitions taken by $\mathcal{P}'_1, \dots, \mathcal{P}'_m$ are valid. Note that, by design, each \mathcal{P}'_i will only output at most one character of the form (δ, q, q', e, g, l) for each $g \in [1, k]$. We assert, if one thread uses a global transition with condition δ , all do, and δ is satisfied. That is,

$$\text{Syncs}(z) \equiv \bigwedge_{1 \leq g \leq k} \bigvee_{\substack{1 \leq e \leq N_{\max} \\ (\delta, \theta, u) \in \Delta_g \\ l \in \{0, 1\}}} \left(\begin{array}{l} \text{Fired}_{(\delta, e, g, l)}(z) \Rightarrow \\ \left(\text{Sync}_{(\delta, e, g, l)}(z) \wedge \text{All Fired}_{(\delta, e, g, l)}(z) \right) \end{array} \right)$$

where $\text{Sync}_{(\delta, e, g, l)}(z)$ is $\delta(z)$ with each atomic state constraints replaced as below.

$$(y_i = q) \equiv \bigvee_{(\delta, q, q', e, g, l)} z_{f(\delta, q, q', e, g, l)} > 0 \quad \text{and} \quad (y_i = q') \equiv \bigvee_{(\delta, q, q', e, g, l)} z_{f(\delta, q, q', e, g, l)} > 0 .$$

Finally, $\text{Fired}_{(\delta, e, g, l)}(z) \equiv \bigvee_{(\delta, q, q', e, g, l)} z_{f(\delta, q, q', e, g, l)} > 0$, and

$$\text{All Fired}_{(\delta, e, g, l)}(z) \equiv \bigwedge_{1 \leq i \leq m} \bigvee_{q, q' \in \mathcal{Q}_i} z_{f(\delta, q, q', e, g, l)} > 0 .$$

We remark upon a pleasant corollary of our main result. Consider a system of pushdown systems communicating only via reversal-bounded counters. Since such a system cannot use any synchronising transitions, all runs are 0-synchronisation-bounded; hence, their reachability problem is in NP.

4 Comparison with Context-Bounded Model Checking

Global synchronisations can be used to model classical context-bounded model checking. We present a simple encoding here. We begin with the definition.

Definition 4 (n-ClPDS). A classical system of communicating pushdown systems with n counters \mathbb{C} is a tuple $(\mathcal{P}_1, \dots, \mathcal{P}_m, G, X)$ where, for all $1 \leq i \leq m$, \mathcal{P}_i is a PDA with n counters $(\mathcal{Q}_i, \Sigma_i, \Gamma, \Delta_i, X)$, X is a finite set of counter variables and $\mathcal{Q}_i = G \times \mathcal{Q}'_i$ for some finite set \mathcal{Q}'_i .

A configuration of a n-ClPDS is a tuple $(g, q_1, w_1, \dots, q_m, w_m, \mathbf{v})$ where $g \in G$ and $(g, q_i) \in \mathcal{Q}_i$ for all i . We have a transition $(g, q_1, w_1, \dots, q_m, w_m, \mathbf{v}) \xrightarrow{\gamma} (g', q'_1, w'_1, \dots, q'_m, w'_m, \mathbf{v}')$ when, for some $1 \leq i \leq m$, we have $((g, q_i), w_i, \mathbf{v}) \xrightarrow{\gamma} ((g', q'_i), w'_i, \mathbf{v}')$ is a transition of \mathcal{P}_i and $q_j = q'_j$ and $w_j = w'_j$ for all $j \neq i$.

A run of \mathbb{C} is a sequence $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \dots \xrightarrow{\gamma_m} c_m$. A k -context-bounded run is a run $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \dots \xrightarrow{\gamma_m} c_m$ that can be divided into k phases C_1, \dots, C_k such that during each C_i only transitions from a unique \mathcal{P}_j are used. By convention, the first phase contains only transitions from \mathcal{P}_1 .

We define an n-SyncPDS simulating any given n-ClPDS. It uses the synchronisations to pass the global component g of the n-ClPDS between configurations of the n-SyncPDS, acting like a token enabling one process to run. Since there are k global synchronisations, the run will be k -context-bounded.

Definition 5. Given a n -ClPDS $\mathbb{C} = (\mathcal{P}_1, \dots, \mathcal{P}_m, G, X)$. Let $\#$ be a symbol not in G . We define from each $\mathcal{P}_i = (Q_i, \Sigma_i, \Gamma, \Delta_i, X)$ with $Q_i = G \times Q'_i$ the pushdown system $\mathcal{P}_i^S = (Q_i^S \cup \{f_i\}, \Sigma_i, \Gamma, \Delta_i^S, X)$ where $Q_i^S = Q'_i \times (G \cup \{\#\})$ and Δ_i^S is the smallest set containing Δ_i and $((g, q), a, tt) \xrightarrow{\varepsilon} (f_i, w, \mathbf{0})$ for all q, a appearing as a head in the final configuration with $g = \#$ or with g also in the final configuration.

Finally, let $\mathbb{C}^S = (\mathcal{P}_1^S, \dots, \mathcal{P}_m^S, \Delta_g, X)$, where $\Delta_g = \{(\delta, tt, \mathbf{0})\}$ such that the formula $\delta(q_1^1, \dots, q_n^1, q_1^2, \dots, q_n^2)$ holds only when there is some $g \in G$ and $1 \leq i \neq j \leq n$ such that

1. $q_i^1 = (g, q)$ and $q_i^2 = (\#, q)$ for some q , and
2. $q_j^1 = (\#, q)$ and $q_j^2 = (g, q)$ for some q , and
3. for all $i' \neq i$ and $i' \neq j$, $q_{i'}^1 = (\#, q)$ and $q_{i'}^2 = (\#, q)$ for some q .

We show in the full version of this paper that an optimised version of this simulation — discussed in Section 5 — is correct. That is, there is a run of \mathbb{C} to the final configuration $(g, q_1, w_1, \dots, q_m, w_m, \mathbf{v})$ iff there is a run of \mathbb{C}^S to $(f_1, w_1, \dots, f_m, w_m, \mathbf{v})$ using the same number of reversals.

5 Optimisations

Our experiments suggest that without further optimisations our reduction from Section 3 is rather impractical. In this section, we provide several optimisations which considerably improve the practical aspect of our reduction. We discuss improving the encoding of the context-bounded model checking, identifying and eliminating “removable” heads from our models, and minimising the size of the CFG produced during the reduction using reachability information. The gist behind our optimisation strategies is to keep the size of the models (pushdown automata, CFG, etc.) as small as possible *throughout* our reduction, which can be achieved by removing redundant objects as early as possible. For the rest of this section, we fix an initial and a final configuration.

Context-Bounded Model Checking. The encoding context-bounded model checking encoding given in Section 4 allows context-switches to occur at any moment. However, we can observe that context-switches only need to occur when global information needs to be up-to-date. This restriction led to improvements in our experiments. We describe the positions where context-switches may occur informally here, and give a formal definition and proof in the long version.

In our restricted encoding, context-switches may occur when an update to global component g occurs; the value of g is tested; an update to the counters occurs; the values of the counters are tested; or the control state of the active thread appears in the final configuration. Intuitively, we delay context-switches

as long as possible without removing behaviours — that is, until the status of the global information affects, or may be affected by, the next transition.

Minimising Communicating Pushdown Systems with Counters. We describe a minimisation technique to reduce the size of the pushdown systems. It identifies heads q, a of the pushdown systems that are *removable*. We collapse pairs of rules passing through the head q, a into a single combined rule. Thus, we build a pushdown system with fewer heads, but the same behaviours. In the following definition, *sink states* will be defined later. Intuitively it means when q is reached, q cannot be changed in one local or global transition.

Definition 6. A head q, a is removable whenever

1. q, a is not the head of the initial or final configuration, and
2. it is not a return location, i.e. there is no rule $(q_1, b_1, \theta) \xrightarrow{\gamma} (q_2, b_2 w', \mathbf{u})$ with a appearing in w' and a does not appear below the top of the stack in the initial configuration, and
3. it is not a loop, i.e. there is no rule $(q, a, \theta) \xrightarrow{\gamma} (q, aw', \mathbf{u})$, and
4. it is not a synchronisation location, i.e. for all $(q_2, b, \theta) \xrightarrow{\gamma} (q_1, aw', \mathbf{u})$ or initial configuration containing q_1 and a , we have either, (i) for all $(\delta, \theta', \mathbf{u}') \in \Delta_g$ and $q_1^1, \dots, q_m^1, q_1^2, \dots, q_m^2$ such that $\delta(q_1^1, \dots, q_m^1, q_1^2, \dots, q_m^2)$ holds we have $q_i^j \neq q_1$ for all i, j , or (ii) q_1 is a sink state, and
5. it is not a counter access location, i.e. there is no rule $(q, a, \theta) \xrightarrow{\gamma} (q', w', \mathbf{u})$ such that θ depends on a counter or \mathbf{u} contains a non-zero entry, and there is no rule $(q', b, \theta) \xrightarrow{\gamma} (q, aw', \mathbf{u})$ such that \mathbf{u} contains a non-zero entry.

Definition 7. A state q is a sink state when for all rules $(q, a, \theta) \xrightarrow{\gamma} (q', w', \mathbf{u})$ we have $q' = q$ and for all $(\delta, \theta', \mathbf{u}') \in \Delta_g$ and $q_1^1, \dots, q_m^1, q_1^2, \dots, q_m^2$ with $q_i^1 = q$ for some i such that $\delta(q_1^1, \dots, q_m^1, q_1^2, \dots, q_m^2)$ holds we have $q_i^2 = q$.

Removable heads can be eliminated by merging rules passing through them. In general, this may increase the number of rules, but in practice it leads to significant reductions (see Table II). We show, in the full version of this paper, that this optimisation preserves behaviours of the systems.

Definition 8. Given a n -SyncPDS with global rules Δ_g and a pushdown system \mathcal{P} with n counters $(\mathcal{Q}, \Sigma, \Gamma, \Delta, X)$ and a removable head q, a , we define $\mathcal{P}_{q,a}$ to be $(\mathcal{Q}, \Sigma, \Gamma, \Delta', X)$ where $\Delta' = \Delta$ if Δ_1 is empty and $\Delta' = \Delta_1 \cup \Delta_2$ otherwise, where

$$\Delta_1 = \left\{ (q_1, b, \theta) \xrightarrow{\gamma_1, \gamma_2} (q_2, w, \mathbf{u}_1 + \mathbf{u}_2) \mid \begin{array}{l} (q_1, b, \theta_1) \xrightarrow{\gamma_1} (q, aw_1, \mathbf{u}_1) \in \Delta \wedge \\ (q, a, \theta_2) \xrightarrow{\gamma_2} (q_2, w_2, \mathbf{u}_2) \in \Delta \wedge \\ \theta = (\theta_1 \wedge \theta_2) \wedge w = w_2 w_1 \end{array} \right\}$$

$$\text{and } \Delta_2 = \Delta \setminus \left(\left\{ \begin{array}{l} (q_1, b, \theta) \xrightarrow{\gamma} (q_2, w', \mathbf{u}) \mid q_1, b = q, a \\ (q_1, b_1, \theta) \xrightarrow{\gamma} (q_2, b_2 w', \mathbf{u}) \mid q_2, b_2 = q, a \end{array} \right\} \cup \right).$$

Minimising CFG Size via Pushdown Reachability Table. Recall that our reduction to existential Presburger formulas from an n-SyncPDS makes use of a standard language-preserving reduction from pushdown automata to context-free grammars (CFGs). Unfortunately, the standard translations from PDAs to CFGs incur a cubic blow-up. More precisely, if the input PDA is $\mathcal{P} = (\mathcal{Q}, \Sigma, \Gamma, \Delta, q^0, F)$, the output CFG has size $O(|\Delta| \times |\mathcal{Q}|^2)$. Our experiments suggest that this cubic blow-up is impractical without further optimisation, i.e., the naive translation failed to terminate within a couple of hours for most of our examples. Note that the complexity of translating from PDA to CFG is very much related to the reachability problem for pushdown systems, for which the optimal complexity is a long-standing open problem (the fastest algorithm [11] to date has complexity $O(n^3 / \log n)$ under certain assumptions).

We will now describe two optimisations to improve the size of the CFG that is produced by our reduction in the previous section, the second optimisation gives better performance (asymptotically and empirically) than the first. Without loss of generality, we assume that: (A1) the PDA empties the stack as it accepts an input word, (A2) the transitions of the input PDA are of the form $(p, a) \xrightarrow{\gamma} (q, w)$ where $p, q \in \mathcal{Q}$, $\gamma \in \Gamma^*$, $a \in \Sigma$, and $w \in \Sigma^*$ with $|w| \leq 2$. [It is well-known that any input PDA can be translated into a PDA in this “normal form” that recognises the same language while incurring only a linear blow-up.] The gist behind both optimisations is to refrain from producing redundant CFG rules by looking at the reachability table for the PDA. Keeping the CFG size low in the first place results in algorithms that are more efficient than removing redundant rules *after* the CFG is produced.

Let us first briefly recall a standard language-preserving translation from PDA to CFG. Given a PDA $\mathcal{P} = (\mathcal{Q}, \Sigma, \Gamma, \Delta, q^0, F)$, we construct the following CFG with nonterminals $N = \{S\} \cup \{A_{p,a,q} : p, q \in \mathcal{Q}, a \in \Sigma\}$, terminals Γ , starting nonterminal S , and the following transitions:

- (1) For each $(p, a) \xrightarrow{\gamma} (q, \epsilon) \in \Delta$, the CFG has $A_{p,a,q} \rightarrow \gamma$.
- (2) For each $(p, a) \xrightarrow{\gamma} (p', b) \in \Delta$ and $q \in \mathcal{Q}$, the CFG has $A_{p,a,q} \rightarrow \gamma A_{p',b,q}$.
- (3) For each $(p, a) \xrightarrow{\gamma} (p', cb) \in \Delta$ and $r, q \in \mathcal{Q}$, the CFG has $A_{p,a,q} \rightarrow \gamma A_{p',c,r} A_{r,b,q}$.
- (4) Add $S \rightarrow A_{q^0, \perp, q_F}$ for each $q_F \in F$.

Note that $A_{p,a,q}$ generates all words that can be output by \mathcal{P} from configuration (p, a) ending in configuration (q, ϵ) . Both of our optimisations refrain from generating: (i) CFG rules of type (2) above in the case when $(p', b) \not\rightarrow^* (q, \epsilon)$, and (ii) CFG rules of type (3) in the case when $(p', c) \not\rightarrow^* (r, \epsilon)$ or $(r, b) \not\rightarrow^* (q, \epsilon)$, and (iii) CFG rules of type (4) in the case when $(q^0, \perp) \not\rightarrow^* (q_F, \epsilon)$.

It remains to describe how to build the reachability lookup table for \mathcal{P} with entries of the form (p, a, q) witnessing whether $(p, a) \rightarrow^* (q, \epsilon)$. The first optimisation achieves this by directly applying the *pre** algorithm for pushdown systems described in [16], which takes $O(|\mathcal{Q}|^2 \times |\Delta|)$ time. This optimisation holds for any input PDA and, hence, does not exploit the structure of the PDA that we generated in the previous section. Our second optimisation improves

the pre^* algorithm for pushdown systems from [16] by exploiting the structure of the PDA generated in the previous section, for which each control state is of the form (p, i, j) , where $i, j \in \mathbb{Z}_{>0}$. The crucial observation is that, due to the PDA rules of type (*) generated from the previous section, the PDA that we are concerned with satisfy the following two properties:

- (P0) $((p, i, j), v) \rightarrow^* ((q, i', j'), w)$ implies $i' \geq i$ and $j' \geq j$.
- (P1) $((p, i, j), v) \rightarrow^* ((q, i', j'), w)$ implies for each $d_1, d_2 \in \mathbb{N}$ that we have $((p, i + d_1, j + d_2), v) \rightarrow^* ((q, i' + d_1, j' + d_2), w)$.
- (P2) for each nonempty $v \in \Sigma^*$: $((p, i, j), av) \rightarrow^* ((q, i', j'), v)$ implies we have $((p, i, j), av) \rightarrow^* ((q, i'', j'), v)$ for each $i'' \geq i'$.

Properties (P0) and (P1) imply it suffices to keep track of the *differences* in the mode indices and context indices in the reachability lookup table, i.e., instead of keeping track of all values $((p, i, j), a) \rightarrow^* ((q, i', j'), \epsilon)$, each entry is of the form (p, a, q, d, d') meaning that $((p, i, j), a) \rightarrow^* ((q, i + d, j + d'), \epsilon)$ for each $i, j \in \mathbb{Z}_{>0}$. Property (P2) implies that if (p, a, q, d, d') is an entry in the table, then so is $(p, a, q, d + i, d')$ for each $i \in \mathbb{N}$. Therefore, whenever p, a, q, d' are fixed, it suffices to *only* keep track of the minimum value d such that (p, a, q, d, d') is an entry in the table. We describe the adaptation of the pre^* algorithm for pushdown systems from [16] for computing the specialised reachability lookup table in the full version. The resulting time complexity for computing the specialised reachability lookup table becomes linear in the number of mode indices.

6 Implementation and Experimental Results

We implemented two tools: Pushdown Translator and SynPCo2Z3.

Pushdown Translator. The Pushdown Translator tool, implemented in C++, takes a program in a simple input language and produces an n-SyncPDS. The language supports threads, boolean variables (shared between threads, global to a thread, or local), shared counters, method calls, assignment to boolean variables, counter increment and decrement, branching and assertions with counter and boolean variable tests, non-deterministic branching, goto statements, locks, output, and while loops. The user can specify the number of reversals and context-switches and specify a constraint on the output performed (e.g. find runs where the number of γ characters output equals the number of γ 's). The full syntax is given in the full version of this paper. The translation uses the context-switch technique presented in Definition 5 and the minimisation technique of Definition 8. Furthermore, the constructed pushdown systems only contain transitions for reachable states of each thread (assuming counter tests always pass, and synchronising transition can always be fired).

SynPCo2Z3. Our second tool SynPCo2Z3 is implemented in SWI-Prolog. The input is an n-SyncPDS, reversal bound r , and synchronisation-bound k . Due to the declarative nature of Prolog, the syntax is kept close to the n-SyncPDS definition. The output is an existential Presburger formulas in SMT-LIB format,

supported by Z3. Moreover, the tool implements all the different translations that have been described in this paper (including appendix) with and without the optimisations described in the previous section. The user may also specify a constraint on the output performed by the input n-SyncPDS.

Experiments. We tested our implementation on several realistic benchmarks. One benchmark concerns the producer-consumer examples (with one producer and one consumer) from [31]. We took two examples from [31]: one uses one counter and is erroneous, wherein both producer and consumer might be both asleep (a deadlock), and the other uses two counters and is correct. The n-SyncPDS models of these examples were hand-coded since they use synchronisations rather than the context-switches of Pushdown Translator.

The remaining benchmarks were adapted from modules found in Linux kernel 3.2.1, which contained list- and memory-management, as well as locks for concurrent access. These modules often provided “register” and “unregister” functions in their API. We tested that, when register was called as many times as unregister, the number of calls to `malloc` was equal to the number of calls to `free`. Furthermore, we checked that the module did not attempt to remove an item from an empty list. In all cases, memory and list management was correct. We then introduced bugs by either removing a call to `free`, or a lock statement. Note the translation from C to our input language was by hand, and an automatic translation is an interesting avenue of future work.

The results are shown in Table II. All tests were run on a 2.8GHz Intel machine with 32GB of RAM. Each benchmark had two threads, two context-switches, one counter and one reversal. The size fields give the total number of pushdown rules in the n-SyncPDS, both before and after removable heads minimisation. Tran. Time gives the time it took to produce the SMT formula, Solve Time is the time taken by Z3 (v. 3.2, Linux build). Each cell contains two entries: the first is for the instance with a bug, the second for the correct instance.

Table 1. Results of experimental runs

File	Size	Min. Size	Tran. Time	Solve Time
<code>prod-cons.c</code>	22/13	-/-	0.8s/1.8s	4.2s/6.8s
<code>api.c</code> (rtl8192u)	654/660	202/208	28s/28s	4m19s/4m32s
<code>af_alg.c</code>	506/528	174/204	18s/21s	10m2s/4m47s
<code>hid-quirks.c</code>	557/559	303/303	47s/47s	18m41s/12m5s
<code>dm-target.c</code>	416/436	254/278	27s/29s	36m43s/10m1s

7 Conclusions and Future Work

We have studied the synchronisation-bounded reachability problem for a class of pushdown systems communicating by shared reversal-bounded counters and global synchronisations. This problem was shown to be NP-complete via an efficient reduction to existential Presburger arithmetic, which can be analysed using fast SMT solvers such as Z3. We have provided optimisation techniques for

the models and algorithms and a prototypical implementation of this reduction and experimented on a number of realistic examples, obtaining positive results.

There are several open problems. For instance, one weakness we would like to address is that we cannot represent data symbolically (using BDDs, for example). This prevents us from being competitive with tools such as Getafix [39] for context-bounded model-checking of pushdown systems without counters.

Furthermore, although we can obtain from the SMT solver a satisfying assignment to the Presburger formula, we would like to be able to construct a complete trace witnessing reachability. Additionally, the construction of a counter-example guided abstraction-refinement loop will require the development of new techniques not previously considered. In particular, heuristics will be needed to decide when to introduce new counters to the abstraction.

We may also consider generalisations of context-bounded analysis such as phase-bounds and ordered multi-stack automata. A further challenge will be to adapt our techniques to dynamic thread creation, where *each thread* has its own context-bound, rather than the system as a whole.

Acknowledgments. We thank EPSRC (EP/F036361 & EP/H026878/1), AMIS (ANR 2010 JCJC 0203 01 AMIS) and VAPF (Fondation de Coopération Scientifique du Campus Paris Saclay) for their support.

References

1. Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of Multi-pushdown Automata Is 2ETIME-Complete. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 121–133. Springer, Heidelberg (2008)
2. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. LMCS 7(4) (2011)
3. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
4. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. Commun. ACM 54, 68–76 (2011)
5. Ball, T., Rajamani, S.K.: Bebop: A Symbolic Model Checker for Boolean Programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
6. Barner, S.: H3 mit gleichheitstheorien. Diploma thesis, TUM (2006)
7. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model-Checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
8. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability Analysis of Multithreaded Software with Asynchronous Communication. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
9. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. SIGPLAN Not. 38(1), 62–73 (2003)

10. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)
11. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: POPL, pp. 159–169 (2008)
12. Dang, Z., Ibarra, O.H., Bultan, T., Kemmerer, R.A., Su, J.: Binary Reachability Analysis of Discrete Pushdown Timed Automata. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 69–84. Springer, Heidelberg (2000)
13. 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)
14. D’Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
15. Esparza, J., Ganty, P.: Complexity of pattern-based verification for multithreaded programs. In: POPL, pp. 499–510 (2011)
16. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient Algorithms for Model Checking Pushdown Systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
17. Esparza, J., Kucera, A., Schwoon, S.: Model checking ltl with regular valuations for pushdown systems. Inf. Comput. 186(2), 355–376 (2003)
18. Ganty, P., Majumdar, R., Monmege, B.: Bounded Underapproximations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 600–614. Springer, Heidelberg (2010)
19. Gurari, E.M., Ibarra, O.H.: The complexity of decision problems for finite-turn multicounter machines. J. Comput. Syst. Sci. 22(2), 220–229 (1981)
20. Hague, M., Lin, A.W.: Model Checking Recursive Programs with Numeric Data Types. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 743–759. Springer, Heidelberg (2011)
21. Heußner, A., Leroux, J., Muscholl, A., Sutre, G.: Reachability analysis of communicating pushdown systems. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 267–281. Springer, Heidelberg (2010)
22. Howell, R.R., Rosier, L.E.: An analysis of the nonemptiness problem for classes of reversal-bounded multicounter machines. J. Comput. Syst. Sci. 34(1), 55–74 (1987)
23. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. J. ACM 25(1), 116–133 (1978)
24. Ibarra, O.H., Su, J., Dang, Z., Bultan, T., Kemmerer, R.A.: Counter machines and verification problems. Theor. Comput. Sci. 289(1), 165–189 (2002)
25. Kahlon, V.: Parameterization as abstraction: A tractable approach to the dataflow analysis of concurrent programs. In: LICS, pp. 181–192 (2008)
26. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural Analysis of Concurrent Programs Under a Context Bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
27. Laroussinie, F., Meyer, A., Petonnet, E.: Counting CTL. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 206–220. Springer, Heidelberg (2010)
28. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: POPL, pp. 283–294 (2011)
29. Mayr, R.: Decidability and Complexity of Model Checking Problems for Infinite-State Systems. PhD thesis, TU-München (1998)
30. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)

31. Wikipedia, http://en.wikipedia.org/wiki/Producer-consumer_problem
32. Qadeer, S.: The Case for Context-Bounded Verification of Concurrent Programs. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 3–6. Springer, Heidelberg (2008)
33. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. TOPLAS (2000)
34. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
35. Sen, K., Viswanathan, M.: Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
36. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic Context-Bounded Analysis of Multithreaded Java Programs. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)
37. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A Java Bytecode Checker Based on Moped. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 541–545. Springer, Heidelberg (2005)
38. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS, pp. 161–170. IEEE Computer Society (2007)
39. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222 (2009)
40. Verma, K.N., Seidl, H., Schwentick, T.: On the Complexity of Equational Horn Clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 337–352. Springer, Heidelberg (2005)

Software Model Checking via IC3

Alessandro Cimatti and Alberto Griggio*

Fondazione Bruno Kessler – IRST
`{cimatti, griggio}@fbk.eu`

Abstract. IC3 is a recently proposed verification technique for the analysis of sequential circuits. IC3 incrementally overapproximates the state space, refuting potential violations to the property at hand by constructing relative inductive blocking clauses. The algorithm relies on aggressive use of Boolean satisfiability (SAT) techniques, and has demonstrated impressive effectiveness.

In this paper, we present the first investigation of IC3 in the setting of software verification. We first generalize it from SAT to Satisfiability Modulo Theories (SMT), thus enabling the direct analysis of programs after an encoding in form of symbolic transition systems. Second, to leverage the Control-Flow Graph (CFG) of the program being analyzed, we adapt the “linear” search style of IC3 to a tree-like search. Third, we cast this approach in the framework of lazy abstraction with interpolants, and optimize it by using interpolants extracted from proofs, when useful.

The experimental results demonstrate the great potential of IC3, and the effectiveness of the proposed optimizations.

1 Introduction

Aaron Bradley [6] has recently proposed IC3, a novel technique for the verification of reachability properties in hardware designs. The technique has been immediately generating strong interest: it has been generalized to deal with liveness properties [5], and to incremental reasoning [7]. A rational reconstruction of IC3, referred to as Property Driven Reachability (PDR), is presented in [12], together with an efficient implementation: an experimental evaluation shows that IC3 is superior to any other single solver used in the hardware model checking competition. See also [23] for an overview.

The technique has several appealing aspects. First, different from bounded model checking, k-induction or interpolation, it does not require unrolling the transition relation for more than one step. Second, reasoning is highly localized to restricted sets of clauses, and driven by the property being analyzed. Finally, the method leverages the power of modern incremental SAT solvers, able to efficiently solve huge numbers of small problems.

In this paper, we investigate the applicability of IC3 to software model checking. We follow three subsequent steps. We first generalize IC3 from the purely Boolean case [6], based on SAT, to the case of Satisfiability Modulo Theory (SMT) [1]. The

* Supported by Provincia Autonoma di Trento and the European Community’s FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 “progetto Trentino”, project ADAPTATION.

characterizing feature of the generalization is the computation of (underapproximations of) the preimage of potential bug states. This allows us to deal with software modeled as a (fully) symbolic transition system, expressed by means of first order formulae.

The second step is motivated by the consideration that the fully symbolic representation does not exploit the control flow graph (CFG) of the program. Thus, we adapt IC3, that is “linear” in nature, to the case of a tree, which is the Abstract Reachability Tree (ART) resulting from the unwinding of the CFG. This technique, that we refer to as TREE-IC3, exploits the disjunctive partitioning of the software, implicit in the CFG.

The third step stems from the consideration that TREE-IC3 can be seen as a form of lazy abstraction with interpolants [18]: the clauses produced by IC3 are in fact interpolants at the various control points of the ART. From this, we obtain another optimization, by integrating interpolation within IC3. With proof-based interpolation, once the path being analyzed is shown to be unfeasible with one SMT call, it is possible to obtain interpolants for each control point, at a low cost. The key problem with interpolation is that the behaviour is quite “unstable”, and interpolants can sometimes diverge. On the other hand, IC3 often requires a huge number of individual calls to converge, and may be computationally expensive, especially in the SMT case, although it rarely suffers from a memory blow-up. The idea is then to obtain clause sets for IC3 from proof-based interpolation, in the cases where this is not too costly.

We carried out a thorough set of experiments, evaluating the merits of the three approaches described above, and comparing with other techniques for software model checking. The results show that the explicit management of the CFG is often superior to a symbolic encoding, and that the hybrid computation of clauses can sometimes yield significant speed-ups. A comparison with other approaches shows that TREE-IC3 can compete with mature techniques such as predicate abstraction, and lazy abstraction with interpolants.

This work has two key elements of novelty. The seminal IC3 [6] and all the extensions we are aware of [12, 7, 15] address the problem for fully symbolic transition systems at the bit level. This paper is the first one to lift IC3 from SAT to SMT, and also the first one to adapt IC3 to exploit the availability of the CFG.

This paper is structured as follows. In Sec. 2 we present some background. In Sec. 3 we describe the SMT generalization of IC3. In Sec. 4 we present TREE-IC3, and in Sec. 5 TREE-IC3+ITP, the hybrid approach using interpolants extracted from proofs. In Sec. 6 we experimentally evaluate the approach. In Sec. 7 we discuss related work. Finally, in Sec. 8 we draw some conclusions and outline lines of future research.

2 Background and Notation

Our setting is standard first order logic. We use the standard notions of theory, satisfiability, validity, logical consequence. We denote formulas with φ, ψ, I, T, P , variables with x, y , and sets of variables with X, Y . Unless otherwise specified, we work on quantifier-free formulas, and we refer to 0-arity predicates as Boolean variables, and to 0-arity uninterpreted functions as (theory) variables. A literal is an atom or its negation. A *clause* is a disjunction of literals, whereas a *cube* is a conjunction of literals. If s is a cube $l_1 \wedge \dots \wedge l_n$, with $\neg s$ we denote the clause $\neg l_1 \vee \dots \vee \neg l_n$, and vice

versa. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, and in disjunctive normal form (DNF) if it is a disjunction of cubes. With a little abuse of notation, we might sometimes denote formulas in CNF $C_1 \wedge \dots \wedge C_n$ as sets of clauses $\{C_1, \dots, C_n\}$, and vice versa. If X_1, \dots, X_n are sets of variables and φ is a formula, we might write $\varphi(X_1, \dots, X_n)$ to indicate that all the variables occurring in φ are elements of $\bigcup_i X_i$. For each variable x , we assume that there exists a corresponding variable x' (the *primed version* of x). If X is a set of variables, X' is the set obtained by replacing each element x with its primed version. Given a formula φ , φ' is the formula obtained by adding a prime to each variable occurring in φ , and $\varphi^{(n)}$ is the formula obtained by adding n primes to each of its variables. Given a theory \mathcal{T} , we write $\varphi \models_{\mathcal{T}} \psi$ (or simply $\varphi \models \psi$) to denote that the formula ψ is a logical consequence of φ in the theory \mathcal{T} . Given a first-order formula φ , we call the *Boolean skeleton* of φ the propositional formula obtained by replacing each theory atom in φ with a fresh Boolean variable.

We represent a program by a *control-flow graph* (CFG). A CFG $A = (L, G)$ consists of a set L of program locations, which model the program counter pc , and a set $G \subseteq L \times \text{Ops} \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to another. The set of variables that occur in operations from Ops is denoted by X . We use first-order formulas for modeling operations: each operation $o \in \text{Ops}$ has an associated first-order formula $T_o(X, X')$ modeling the effect of performing the operation o . A *program* $\Pi = (A, \text{pc}_0, \text{pc}_E)$ consists of a CFG $A = (L, G)$, an initial program location $\text{pc}_0 \in L$ (the program entry), and a target program location $\text{pc}_E \in L$ (the error location). A *path* π is a sequence $(\text{pc}_0, o_0, \text{pc}_1), (\text{pc}_1, o_1, \text{pc}_2), \dots, (\text{pc}_{n-1}, o_{n-1}, \text{pc}_n)$, representing a syntactical walk through the CFG. The path π is *feasible* iff the formula $\bigwedge_i T_{o_i}^{(i)}$ is satisfiable. When π is not feasible, we say it is *spurious*. A program is *safe* when all the paths leading to pc_E are not feasible.

Given a program Π , an *abstract reachability tree* (ART) for Π is a tree \mathcal{A} over (V, E) such that: (i) V is a set of triples (pc, φ, h) , where $\text{pc} \in L$ is a location in the CFG of Π , φ is a formula over X , and $h \in \mathbb{N}$ is a unique identifier; (ii) the root of \mathcal{A} is $(\text{pc}_0, \top, 1)$; (iii) for every non-leaf node $v \stackrel{\text{def}}{=} (\text{pc}_i, \varphi, h) \in V$, for every control-flow edge $(\text{pc}_i, o, \text{pc}_j) \in G$, v has a child node (pc_j, ψ, k) such that $\varphi \wedge T_{op} \models \psi'$ and $k > h$. In what follows, we might denote with $\text{pc}_i \rightsquigarrow \text{pc}_j$ any path in an ART from a node $(\text{pc}_i, \varphi, h)$ to a descendant node (pc_j, ψ, k) . Intuitively, an ART represents an unwinding of the CFG of a program performed in an abstract state space. If $v \stackrel{\text{def}}{=} (\text{pc}_i, \varphi, h)$ is a node, φ is the *abstract state formula* of v . A node $v_1 \stackrel{\text{def}}{=} (\text{pc}_i, \psi, k)$ in an ART \mathcal{A} is *covered* if either: (i) there exists another node $v_2 \stackrel{\text{def}}{=} (\text{pc}_i, \varphi, h)$ in \mathcal{A} such that $h < k$, $\psi \models \varphi$, and v_2 is not itself covered; or (ii) v_1 has a proper ancestor for which (i) holds. \mathcal{A} is *complete* if all its leaves are either covered or their abstract state formula is equivalent to \perp . \mathcal{A} is *safe* if and only if it is complete and, for all nodes $(\text{pc}_E, \varphi, h) \in V$, $\varphi \models \perp$. If a program Π has a safe ART, then Π is safe [16, 18].

Given a set X of (state) variables, a *transition system* S over X can be described symbolically with two formulas: $I_S(X)$, representing the initial states of the system, and $T_S(X, X')$, representing its transition relation. Given a program Π , a corresponding transition system S_{Π} can be obtained by encoding symbolically the CFG (L, G)

of Π . This can be done by: (i) adding one special element x_{pc} , with domain L , to the set X of variables; (ii) setting $I_{S_\Pi} \stackrel{\text{def}}{=} (x_{\text{pc}} = \text{pc}_0)$; and (iii) setting $T_{S_\Pi} \stackrel{\text{def}}{=} \bigvee_{(\text{pc}_i, op, \text{pc}_j) \in G} (x_{\text{pc}} = \text{pc}_i) \wedge T_{op} \wedge (x'_{\text{pc}} = \text{pc}_j)$.

Given S_Π , the safety of the program Π can be established by proving that all the reachable states of S_Π are a subset of the states symbolically described by the formula $P \stackrel{\text{def}}{=} \neg(x_{\text{pc}} = \text{pc}_E)$. In this case, we say that S_Π satisfies the invariant property P .

3 IC3 with SMT

High-Level Description of IC3. Let X be a set of Boolean variables, and let S be a given Boolean transition system described symbolically by $I(X)$ and $T(X, X')$. Let $P(X)$ describe a set of good states. The objective is to prove that all the reachable states of S are good. (Conversely, $\neg P(X)$ represents a set of “bad” states, and the objective is to show that there exists no sequence of transitions from states in $I(X)$ to states in $\neg P(X)$.) The IC3 algorithm tries to prove that S satisfies P by finding a formula $F(X)$ such that: (i) $I(X) \models F(X)$; (ii) $F(X) \wedge T(X, X') \models F(X')$; and (iii) $F(X) \models P(X)$.

In order to construct F , which is an inductive invariant, IC3 maintains a sequence of formulas (called *trace*, following [12]) $F_0(X), \dots, F_k(X)$ such that:

- $F_0 = I$;
- for all $i > 0$, F_i is a set of clauses;
- $F_{i+1} \subseteq F_i$ (thus, $F_i \models F_{i+1}$);
- $F_i(X) \wedge T(X, X') \models F_{i+1}(X')$;
- for all $i < k$, $F_i \models P$;

The algorithm proceeds incrementally, by alternating two phases¹: a blocking phase, and a propagation phase. In the *blocking* phase, the trace is analyzed to prove that no intersection between F_k and $\neg P(X)$ is possible. If such intersection cannot be disproved on the current trace, the property is violated and a counterexample can be reconstructed. During the blocking phase, the trace is enriched with additional clauses, that can be seen as strengthening the approximation of the reachable state space. At the end of the blocking phase, if no violation is found, $F_k \models P$.

The *propagation* phase tries to extend the trace with a new formula F_{k+1} , moving forward the clauses from preceding F_i . If, during this process, two consecutive elements of the trace (called *frames*) become identical (i.e. $F_i = F_{i+1}$), then a fixpoint is reached, and IC3 can terminate with F_i being an inductive invariant proving the property.

Let us now consider the lower level details of IC3. For $i > 0$, F_i represents an over-approximation of the states of S reachable in i transition steps or less. The distinguishing feature of IC3 is that such sets of clauses are constructed incrementally, starting from cubes representing sets of states that can reach a bad state in zero or more

¹ We follow the formulation of IC3 given in [12], which is slightly different from the original one of Bradley given in [6]. Moreover, for brevity we have to omit several important details, for which we refer to the two papers cited above.

```

bool IC3-prove( $I, T, P$ ):
1. trace = [ $I$ ] # first elem of trace is init formula
2. trace.push() # add a new frame to the trace
3. while True:
   # blocking phase
   4. while there exists a cube  $c$  s.t. trace.last()  $\wedge T \wedge c$  is satisfiable and  $c \models \neg P$ :
   5.   recursively block the pair  $(c, \text{trace.size}() - 1)$ 
   6.   if a pair  $(p, 0)$  is generated:
      7.     return False # counterexample found
   # propagation phase
   8. trace.push()
   9. for  $i = 1$  to trace.size() - 1:
      10.   for each clause  $c \in \text{trace}[i]$ :
         11.     if  $\text{trace}[i] \wedge c \wedge T \wedge \neg c'$  is unsatisfiable:
            12.       add  $c$  to  $\text{trace}[i+1]$ 
         13.     if  $\text{trace}[i] == \text{trace}[i+1]$ :
            14.       return True # property proved

```

Fig. 1. High-level description of IC3 (following [12])

transition steps. More specifically, in the blocking phase, IC3 maintains a set of pairs (s, i) , where s is a cube representing a set of states that can lead to a bad state, and $i > 0$ is a position in the current trace. New clauses to be added to (some of the frames in) the current trace are derived by (recursively) proving that a set s of a pair (s, i) is unreachable starting from the formula F_{i-1} . This is done by checking the satisfiability of the formula:

$$F_{i-1} \wedge \neg s \wedge T \wedge s'. \quad (1)$$

If (1) is unsatisfiable, and s does not intersect the initial states I of the system, then $\neg s$ is *inductive relative to* F_{i-1} , and IC3 strengthens F_i by adding $\neg s$ to it², thus *blocking* the bad state s at i . If, instead, (1) is satisfiable, then the overapproximation F_{i-1} is not strong enough to show that s is unreachable. In this case, let p be a cube representing a subset of the states in $F_{i-1} \wedge \neg s$ such that all the states in p lead to a state in s' in one transition step. Then, IC3 continues by trying to show that p is not reachable in one step from F_{i-2} (that is, it tries to block the pair $(p, i - 1)$). This procedure continues recursively, possibly generating other pairs to block at earlier points in the trace, until either IC3 generates a pair $(q, 0)$, meaning that the system does not satisfy the property, or the trace is eventually strengthened so that the original pair (s, i) can be blocked. Figure 1 reports the pseudo-code for the full IC3 algorithm, including more details on the propagation phase.

Extension to SMT. In its original formulation, IC3 works on finite-state systems, with Boolean state variables and propositional logic formulas, using a SAT solver as its reasoning engine. However, for modeling programs it is often more convenient to reason at a higher level of abstraction, using (decidable) fragments of first-order logic and SAT modulo theories (SMT).

² In fact, $\neg s$ is actually *generalized* before being added to F_i . Although this is quite important for the effectiveness of IC3, here for simplicity we shall not discuss this.

Most of the machinery of IC3 can be lifted from SAT to SMT in a straightforward way, by simply replacing the underlying SAT engine with an SMT solver. From the point of view of IC3, in fact, it is enough to reason at the level of the Boolean skeleton of formulas, simply letting the SMT solver cope with the interpretation of the theory atoms. There is, however, one crucial step in which IC3 must be made theory-aware, as reasoning at the Boolean-skeleton level does not work. This happens in the blocking phase, when trying to block a pair (s, i) . If the formula $\text{I}\Box$ is satisfiable, then a new pair $(p, i - 1)$ has to be generated such that p is a cube in the *preimage* of s wrt. T . In the purely-Boolean case, p can be obtained from the model μ of $\text{I}\Box$ generated by the SAT solver, by simply dropping the primed variables occurring in $\mu\Box$.³ This cannot be done in general in the first-order case, where the relationship between the current state variables X and their primed version X' is encoded in the theory atoms, which in general cannot be partitioned into a primed and an unprimed set.

A first (and rather naïve) solution would be to consider the theory model for the state variables X generated by the SMT solver. However, for infinite-state systems this would lead IC3 to exclude only a single point at a time. This will most likely be impractical: being the state space infinite, there would be a high chance that the blocking phase will diverge.

For theories admitting quantifier elimination, a better alternative is to compute an exact preimage of s . This means to existentially quantify the variables X' in $\text{I}\Box$, eliminate the quantifiers, and then convert the result in DNF. This will generate a set of cubes $\{p_j\}_j$ which in turn generate a set of pairs $\{(p_j, i - 1)\}_j$ to be blocked at $i - 1$. The drawback of the second solution is that for many important theories, even when it is possible, quantifier elimination may be a very expensive operation.

We notice that the two solutions above are just the two extremes of a range of possibilities: in fact, any procedure that is able to compute an under-approximation of the exact preimage can be used. Depending on the theory, several trade-offs between precision and computational cost can be explored, ranging from single points in the state space to a precise enumeration of all the cubes in the preimage. In what follows, we shall assume that we have a procedure APPROX-PREIMAGE for computing such under-approximations, and present our algorithms in a general context. We shall discuss our current implementation, which uses the theory of Linear Rational Arithmetic, in §6.

Discussion. We conclude this Section by pointing out that the ideas underlying IC3 are nontrivial even in the Boolean case. At a very high level, the correctness is based on the invariants ensured by the blocking and propagation phases. Termination follows from the finiteness of the state space being analyzed, and from the fact that at each step at least one more new state is explored. A more in depth justification is out of the scope of this paper. The interested reader is referred to [6][23][2].

In the case of SMT, we notice that the invariants of the traces also hold in the SMT case, so that the argument for the finite case can be applied. This ensures partial correctness. On the other hand, the reachability problem being undecidable for infinite-state transition systems, it is impossible to guarantee termination. This might be due to the

³ For efficiency, the result has to be generalized by dropping irrelevant variables, but this is not important for the discussion here.

failure in the blocking phase to eliminate all the counterexamples, for the given trace length, or to the failure to reach a fixpoint in the propagation phase.

4 Tree-Based IC3

We now present an adaptation of IC3 from symbolic transition systems to a CFG-represented program. The search proceeds in an “explicit-symbolic” approach, similarly to the lazy abstraction approach [16]. The CFG is unwound into an ART (Abstract Reachability Tree), following a DFS strategy. Each node of the tree is associated with a location, and a set of clauses.

The algorithm starts by finding an abstract path to the error location. Then, it applies a procedure that mimics the blocking phase of IC3 on the sets of clauses of the path.

There are three important differences. First, the clauses associated to a node are implicitly conditioned to the corresponding control location: the clause $\neg(x_{\text{pc}} = \text{pc}_i) \vee c$ in the fully symbolic setting simply becomes c in a node associated with control location pc_i . This also means that the logical characterization of a node being unreachable, expressed by the clause $\neg(x_{\text{pc}} = \text{pc}_E)$ in the fully symbolic setting, is now the empty clause. Second, in each formula T_i characterizing a transition, the start and end control locations are not explicitly represented, but rather implicitly assumed. Finally, the most important difference is in the inductiveness check performed when constructing the IC3 trace. When checking whether a cube c is blocked by a set of clauses F_{i-1} , we cannot use the relative inductiveness check of (1). This is because that would not be sound in our setting, since we are using different transition formulas T_i at different i steps (corresponding to the edge formulas in the abstract error path). Therefore, we replace (1) with the weaker check

$$F_{i-1} \wedge T_{i-1} \models \neg c' \quad (2)$$

which allows us to construct a correct ART (satisfying points (i)–(iii) of the definition on page 279.) We observe that, because of this difference, the requirement that $F_{i+1} \subseteq F_i$ is not enforced in TREE-IC3.

With this adaptation, the blocking phase tries to produce the clauses necessary to refute the abstract path. When the blocking phase is successful, it must generate an empty clause at some point. In case of failure to refute the path, the property is violated and a counterexample is produced.⁴ If sufficient information can be devised to refute the abstract path to the error location, the algorithm backtracks to the deepest node that is not inconsistent (i.e. is not associated with the empty clause). The pseudo-code of this modified blocking phase, which we call TREE-IC3-BLOCK-PATH, is reported in Figure 2.

Then, a new node is selected and expanded, with a process that is similar in nature to the forward propagation phase of IC3. For each expanded node, the clauses of the ancestor are tested for forward propagation, in order to ensure the invariant that the clauses of an abstract node overapproximate the image of the predecessor clauses. More specifically, for each clause c , we check whether $F_i \wedge (x_{\text{pc}} = \text{pc}_i) \rightarrow c \wedge T_{op}$ entails $(x'_{\text{pc}} = \text{pc}_{i+1}) \rightarrow c'$.

⁴ The counterexample has exactly the same length as the abstract path. This is a key difference with respect to the case of the fully symbolic IC3.

```

procedure TREE-IC3-BLOCK-PATH ( $\pi \stackrel{\text{def}}{=} (\text{pc}_0, \top, 1) \rightsquigarrow \dots (\text{pc}_i, \varphi_i, \cdot) \dots \rightsquigarrow (\text{pc}_E, \varphi_n, \cdot)$ ):  

    #  $T_1 \dots T_{n-1}$  are the edge formulas of  $\pi$   

    # initialize the trace with the clauses attached to the nodes in  $\pi$   

1.  $F = [\top, \dots, \varphi_i, \dots, \varphi_{n-1}]$   

2. while not exists  $j$  in  $1 \dots n - 1$  s.t.  $F[j] \wedge T_j \models \perp$ :  

3.    $q = []$   

4.   for each bad in APPROX-PREIMAGE ( $\varphi_{n-1} \wedge T_{n-1}$ ):  

5.      $q.push((\text{bad}, n - 1))$  # bad is a cube in the preimage of  $T_{n-1}$   

6.   while q is not empty:  

7.      $c, j = q.top()$   

8.     if  $j = 0$ : compute and return a counterexample trace #  $\pi$  is a feasible error trace  

9.     if  $F[j - 1] \wedge T_{j-1} \models \neg c'$ :  

10.        $q.pop()$  # c is blocked, discard the proof obligation  

11.        $g = \text{generalization of } \neg c \text{ s.t. } F[j - 1] \wedge T_{j-1} \models g'$   

12.        $F[j] = F[j] \wedge g$   

13.     else:  

14.       for each p in APPROX-PREIMAGE ( $F[j - 1] \wedge T_{j-1} \wedge c'$ ):  

15.          $q.push((p, j - 1))$   

16.   return  $F$  #  $\pi$  is blocked

```

Fig. 2. Modified blocking phase of TREE-IC3 for refuting a spurious error path

A significant difference with respect to IC3 is in the way the fix point is handled. In IC3 the fix point is detected globally, by comparing two subsequent formulae in the trace. Here, as standard in lazy abstraction, we close each path of the ART being generated.

Whenever a new node v' is expanded, it is checked against previously generated nodes v having the same location. If the set of states of v' is contained in the states of some previously generated node v , then v' is covered, and it can be closed.⁵

In order to maximize the probability of coverage, the IC3-like forward propagation phase is complemented by another form of forward propagation: whenever a loop is encountered (i.e. the node v' being expanded has the same location of one of its ancestors v), then each of the clauses of v is tested to see if it also holds in v' . Let v, v_1, \dots, v_k, v' be the path from v to v' . For each clause c in v , we check if the symbolic encoding of the path $v \rightsquigarrow v'$, strengthened with the clauses in each v_i , entails c in v' .

It is easy to see that this may result in a stronger set of clauses for v' , because the analysis is carried out on the concrete path from v to v' , that retains all the available information. Simple forward propagation would not be able to achieve the same result, because of the limited strength of the clauses on the intermediate nodes v_i . Intuitively, this means that the clauses in v_i may be compatible with (too weak to block) the paths that violate the clauses of v that also hold in v' . Thus, simply strengthening v' would break the invariant that, in each node, the abstract state formula overapproximates the image of the abstract state formula of its parent node (point (iii) in the definition of ART, §2). In order to restore the situation, the v_i nodes must be strengthened. Let $C_{v, v'}$

⁵ In fact, it is also required that there will be no cycles in the covering-uncovering interplay. This requirement is a bit technical, and discussed in detail in [18]. In the definition of covered node, in §2 identifiers to nodes are intended to enforce this requirement.

ART UNWINDING: if $v \stackrel{\text{def}}{=} (\text{pc}_i, \varphi, h)$ is an uncovered leaf: for all edges $(\text{pc}_i, op, \text{pc}_j)$ in the CFG: add $v_j \stackrel{\text{def}}{=} (\text{pc}_j, \top, k)$ with $k > h$ as a child of v in the ART	NODE COVERING: if $v_1 \stackrel{\text{def}}{=} (\text{pc}_i, \psi, k)$ is uncovered, and there exists $v_2 \stackrel{\text{def}}{=} (\text{pc}_i, \varphi, h)$ with $k > h$ and $\psi \models \varphi$, then: mark v_1 as covered by v_2 uncover all the nodes $v_j \stackrel{\text{def}}{=} (\text{pc}_i, \psi_j, k_j)$ covered by v_1
PATH BLOCKING: if $v_E \stackrel{\text{def}}{=} (\text{pc}_E, \varphi, h)$ is a leaf with $\varphi \not\models \perp$: apply TREE-IC3-BLOCK-PATH (Fig. 2) to the ART path $\pi \stackrel{\text{def}}{=} (\text{pc}_0, \top, 1) \rightsquigarrow v_E$ if IC3 returns a counterexample: return UN- SAFE otherwise: let F_1, \dots, F_E be the sets of clauses computed by IC3 for the formulas $T_{op_1}, \dots, T_{op_E}$ of π for each node $v_i \stackrel{\text{def}}{=} (\text{pc}_i, \varphi_i, h_i) \in \pi$, for each clause c_j in the corresponding F_i : if $\varphi_i \not\models c_j$, then: add c_j to φ_i uncover all the nodes covered by v_i	STRENGTHENING: let $v_1 \stackrel{\text{def}}{=} (\text{pc}_i, \varphi, h_1)$ and $v_2 \stackrel{\text{def}}{=} (\text{pc}_k, \psi, h_2)$ be two uncovered nodes s.t. there is a path $\pi \stackrel{\text{def}}{=} v_1 \rightsquigarrow v_2$, and let $\phi_\pi \stackrel{\text{def}}{=} \bigwedge_{j=0}^n T_{op_j}^{(j)}$ be the formula for π let $C_{v_1, v_2} = \emptyset$ for each $c_j \in \varphi$: if $\psi \not\models c_j$ and $\varphi^{(0)} \wedge \phi_\pi \models c_i^{(n)}$: add c_j to C_{v_1, v_2} if $C_{v_1, v_2} \neq \emptyset$: refute $\neg C_{v_1, v_2}$ using TREE-IC3-BLOCK-PATH along π for each node $v_j \stackrel{\text{def}}{=} (\text{pc}_j, \varphi_j, h_j) \in \pi$: add all the clauses $c \in F_j$ computed by IC3 s.t. $\varphi_j \not\models c$ if φ_j changes, uncover all the nodes covered by v_j add C_{v_1, v_2} to ψ , and uncover all the nodes covered by v_2

Fig. 3. High-level description of the basic building blocks of TREE-IC3

be the set of clauses of v that also hold in v' . Before adding $C_{v, v'}$ to v' , we strengthen the v_i nodes with the information necessary to block the violation of $C_{v, v'}$ in v' . This is done by “tricking” the blocking phase, using the negation of $C_{v, v'}$ as conjecture: the clauses deduced in the process of refuting $\neg C_{v, v'}$ can be added to strengthen each v_i . After this, $C_{v, v'}$ is added to v' .

Notice that whenever a node v is strengthened, then each node v' that had been covered by v must be re-opened. In fact, after the strengthening, the set of states of v shrinks, thus the set of states of v' , that was previously covered, might no longer be contained.

A high-level view of the basic steps of TREE-IC3 is reported in Figure 3. We shall describe the actual strategy that we have implemented for applying these steps in §6. (Notice that the forward propagation that is performed when a node is expanded is just a special case of the more general strengthening procedure, in which the path between the two nodes involved consists of a single edge, and as such does not require a call to IC3 for strengthening the intermediate nodes.)

Comparison with IC3. When the fully symbolic IC3 analyzes a program (in form of symbolic transition system), some literals represent the location in the control flow that is “active”. This information, that is implicit in the position in the ART, becomes direct part of the clauses. There is the possibility for clauses to be present at frames where the corresponding location can not be reached, and that are thus irrelevant. Another advantage of the TREE-IC3 approach is that the program is disjunctively partitioned,

transition by transition, and thus the SMT solver is manipulating simpler and smaller formulae. On the other hand, the symbolic representation gives the ability to implicitly “replicate” the same clause over many control locations – in particular, when no control location is relevant in the clause, it means that it holds for all the control locations. Moreover, using a symbolic representation of the program as a transition formula allows to exploit relative inductiveness, which is crucial for the performance of the original IC3 (on hardware designs). As already mentioned above, relative inductiveness cannot be directly applied in our setting, because we use a disjunctively-partitioned representation. In our experiments (§6), we show that the benefits of a CFG-guided exploration significantly outweigh this drawback in the verification of sequential programs.

5 Hybrid Tree IC3

It can be observed that the sequence of sets of clauses generated by the Tree-based IC3 for refuting a spurious abstract error path can be seen as an *interpolant* for the path, in the sense used by McMillan in his “lazy abstraction with interpolants” algorithm [18].⁶ Recalling the definition of [18], given a sequence of formulas $\Gamma \stackrel{\text{def}}{=} \varphi_1, \dots, \varphi_n$, an interpolant is a sequence of formulas I_0, \dots, I_n such that: (i) $I_0 \equiv \top$ and $I_n \equiv \perp$; (ii) for all $1 \leq i \leq n$, $I_{i-1} \wedge \varphi_i \models I_i$; (iii) for all $1 \leq i \leq n$, I_i contains only variables that are shared between $\varphi_1 \wedge \dots \wedge \varphi_i$ and $\varphi_{i+1} \wedge \dots \wedge \varphi_n$. Consider now a program path $\text{pc}_0 \rightsquigarrow \text{pc}_n$, and its corresponding sequence of edge formulas $T_{op_1}, \dots, T_{op_n}$ (where T_{op_i} is the formula attached to the edge $(\text{pc}_{i-1}, op_i, \text{pc}_i)$). Then, it is easy to see that the trace F_0, \dots, F_n generated by IC3 in refuting such path immediately satisfies points (i) and (ii) above by definition, and, if we consider the sequence $T_{op_1}^{(0)}, \dots, T_{op_n}^{(n-1)}$, then $F_0^{(0)}, \dots, F_n^{(n)}$ satisfies also point (iii).

Under this view, the TREE-IC3 algorithm described in the previous section can be seen as an instance of the lazy abstraction with interpolants algorithm of [18], in which however interpolants are constructed in a very different way. In the algorithm of [18], interpolants are constructed from proofs of unsatisfiability generated by the SMT solver in refuting spurious error paths; as such, the generated interpolants might have a complex Boolean structure, which depends on the structure of the proof generated by the SMT solver. Moreover, they are typically large and possibly very redundant. In the iterative process of expanding and refining ART nodes, it is often the case that interpolants become larger and larger, causing the algorithm to diverge. In fact, in our experiments we have seen several cases in which the interpolant-based algorithm quickly runs out of memory. On the other hand, when the interpolants are “good”, the algorithm is quite fast, since interpolants can be quickly generated using a single call to the SMT solver for each spurious error path.

Consider now the case of TREE-IC3. Here, the interpolants generated, being sets of clauses, have a very regular Boolean structure, and experiments have shown that they are often more compact than those generated from proofs, and as such do not cause blow-ups in memory. Furthermore, another important advantage of having interpolants (that is, abstract state formulas in the ART) in the form of sets of clauses is that this allows to perform strengthening of nodes (see §4) at the level of granularity of individual

⁶ In fact, a similar observation has been done already for the fully-symbolic IC3 [12, 23].

clauses. In [18], strengthening (called “forced covering” there) is an “all or nothing” operation: either the whole abstract state formula φ_P holds at a descendant node φ_N , or no strengthening is performed. As our experimental evaluation in §6 will show, the capability of performing clause-by-clause strengthening is very important for performance.

A drawback of TREE-IC3 is that the construction of the interpolants is typically more expensive than with the proof-based approach, since it requires many (albeit simpler) calls to the SMT solver for each spurious path, and it also requires many potentially-expensive calls to the APPROX-PREIMAGE procedure needed for generalizing IC3 to SMT (see §3). As a solution, we propose a hybrid approach that combines TREE-IC3 with proof-based interpolant generation, in order to get the benefits of both. The main idea of this new algorithm, which we call TREE-IC3+ITP, is that of generating the sets of clauses in the trace of TREE-IC3 starting from the proof-based interpolants, when such interpolants are “good”. More specifically, given an abstract error path $\text{pc}_0 \rightsquigarrow \text{pc}_E$, before invoking IC3 on it, we generate an interpolant I_0, \dots, I_n (for the corresponding edge formulas $T_{\text{op}_1}, \dots, T_{\text{op}_n}$) with the efficient proof-based procedures available in interpolating SMT solvers (see e.g. [9]); then, we try to generate clauses from each I_i by converting them to CNF, using an *equivalence-preserving procedure* (and not, as usual, a satisfiability-preserving one), aborting the computation if this process generates too many clauses. Only when this procedure fails, we fall back to generating sets of clauses with the more expensive IC3. This allows us to keep the performance advantage of the proof-based interpolation method when the generated interpolants are “good”, while still benefiting from the advantages of a clause-based representation of abstract states outlined above. Despite its simplicity, in fact, this hybrid algorithm turns out to be quite effective in practice, as our experiments in the next section show.

6 Implementation and Experiments

We have implemented the algorithms described in the previous sections on top of the MATHSAT5 SMT solver [14] and the KRATOS software model checker [8]. In this section, we experimentally evaluate their performance.

6.1 Implementation Details

Generalization of IC3 to SMT. Our current implementation uses the theory of Linear Rational Arithmetic (LRA) for modeling program operations. LRA is well supported by MATHSAT5, which implements efficient algorithms for both satisfiability checking and interpolation modulo this theory [119]. Moreover (and more importantly), using LRA allows us to implement a simple and not-too-expensive APPROX-PREIMAGE procedure for computing under-approximations of preimages, as required for generalizing IC3 to SMT (see §3). Given a bad cube s and a transition formula $T(X, X')$, the exact preimage of s wrt. T can be computed by converting $s' \wedge T$ to a DNF $\bigvee_i m_i$ and then projecting each of the cubes m_i over the current-state variables X : $\bigvee_i \exists X'.(m_i)$. Then, an under-approximation can be constructed by simply picking only a subset of

the projections of the cubes m_i of the DNF. In our implementation, we use the All-SMT-based algorithm of [20] to construct the DNF lazily, and in order to keep the cost of the computation relatively low we under-approximate by simply stopping after the first cube.

Implementation of IC3. In general, our implementation of IC3 follows the description given in [12] (called PDR there). In order to be implemented efficiently, IC3 requires a very tight integration with the underlying SAT (or SMT) solver, and the details of such integration are sometimes crucial for performance. Therefore, here we precisely outline the differences wrt. the description given in [12]. In particular, besides the obvious one of using an SMT solver instead of a SAT solver, the two main differences are:

- For simplicity, we use a single solver rather than a different solver per frame, as suggested in [12]. Moreover, since MATHSAT5 supports both an incremental interface, through which clauses added and removed in a stack-based manner, and an assumptions-based interface, we use a mixture of both for efficiently querying the solver: we use assumptions for activating and deactivating the clauses of the initial states, transition relation, bad states and those of the individual frames, as described in detail in [12], whereas we use the push/pop interface for temporarily adding a clause to the solver for checking whether such clause is inductive (relative to the previously-generated ones). This allows us to avoid the need of periodically cleaning old activation literals as described in [2].
- For reducing bad cubes that must be blocked during the execution of IC3, we exploit the *dual-rail encoding* typically used in Symbolic Trajectory Evaluation [22]. We do not apply ternary simulation through the transition relation, as suggested in [12] for the Boolean case, as we found the former to be much more efficient than the latter. This is possibly because the data structures that we use for representing formulas are relatively naïve and inefficient.

Implementation of Tree-Based IC3. We adopt the “Large-Block” encoding [3] of the control-flow graph of the program under analysis, which collapses loop-free subparts of the original CFG into a single edge, in order to take full advantage of the power of the underlying SMT solver of efficiently reasoning with disjunctions. Currently, we do not handle pointers or recursive functions, and we inline all the function calls so as to obtain a single CFG.

For the construction of the abstract reachability tree, we adopt the “DFS” strategy described by McMillan in [18]. When constructing the ART, we apply strengthening systematically to each uncovered node n which has a proper ancestor p tagged with the same program location, by adding to n all the clauses of p that hold after the execution of the path $p \rightsquigarrow n$, and strengthening the intermediate nodes accordingly.

Finally, in the “hybrid” version, we use a threshold on the size of the CNF conversion for deciding whether to use interpolants for computing the sets of clauses for refuting a spurious path: we compute the sequence of interpolants, and we try to convert them to CNF, aborting the process when the formulas become larger than k times the size of the interpolants (k being a configurable parameter set to 5 in the experiments)⁷.

⁷ We remark that here we need an *equivalent*, and not just equisatisfiable, CNF representation. Therefore, such conversion might result in an exponential blow-up in the size of the formula.

6.2 Benchmarks and Evaluation

For our evaluation, we use a set of 98 benchmark C programs from the literature, originating from different domains (e.g. device drivers, communication protocols, SystemC designs, and textbook algorithms), most of which have been used in several previous works on software model checking. The set includes the benchmarks used in the first software verification competition (<http://sv-comp.sosy-lab.org>) that can be handled by our implementation. About one third of the programs contain bugs.

All the benchmarks, tools and scripts needed for reproducing the experiments are available at <http://es.fbk.eu/people/griggio/papers/cav12-ic3smt.tar.bz2>. The experiments have been run on a Linux machine with a 2.6GHz CPU, using a time limit of 1200 seconds and a memory limit of 2GB.

For our evaluation, we tested the following algorithms/configurations:

IC3 is the fully symbolic version of IC3, in which the CFG is encoded symbolically in the transition relation using an auxiliary variable representing the program counter⁸.

TREE-IC3 is the CFG-based version of IC3, as described in §4.

TREE-IC3+ITP is the hybrid algorithm of §5 in which “good” interpolants are used for computing sets of inductive clauses;

TREE-IC3+ITP-MONO is a variant of TREE-IC3+ITP in which strengthening of nodes is performed “monolithically” by checking whether *all* the clauses of an ancestor node p hold at a descendant node n , instead of checking the clauses individually. This configuration mimics the forced coverage procedure applied in the lazy abstraction with interpolants algorithm of [18];

TREE-ITP is an implementation of the lazy abstraction with interpolants algorithm of [18];

KRATOS is an implementation of lazy predicate abstraction with interpolation-based refinement, the default algorithm used by the KRATOS software model checker [8]. (We recall that the difference with TREE-ITP is that in standard lazy predicate abstraction interpolants are used only as a source of new predicates, and abstract states are computed using Boolean abstraction rather than using interpolants directly.)

All the implementations use the same front-end for parsing the C program and computing its CFG, and they all use the same SMT solver (MATHSAT5 [14]) as a back-end reasoning engine for all satisfiability checks and interpolation queries. Moreover, all the tree-based algorithms use the same depth-first strategy for constructing the ART, and all the IC3-based ones use the same settings for IC3. This makes it possible to compare the merits of the various algorithms (over the benchmark instances) without being affected by potential differences in the implementation of other parts of the systems which are orthogonal to the evaluation.

Moreover, in addition to comparing the different algorithms within the same implementation framework, we also compared our best algorithm with the following software model checkers:

⁸ More precisely, we encode the program counter variable using $\lceil \log_2 n \rceil$ Boolean variables, where n is the number of locations in the CFG.

CPACHECKER [4], which uses an algorithm based on lazy predicate abstraction [6] (like KRATOS). CPACHECKER was the winner of the first software verification competition⁹.

WOLVERINE [17], an implementation of the lazy abstraction with interpolants algorithm [18] (like TREE-ITP).¹⁰

6.3 Results

The results of the evaluation are summarized in Figure 4. The scatter plots in the top row show the comparisons of the various configurations of IC3 proposed in the previous Sections: first, we compare the fully-symbolic IC3 with TREE-IC3, in order to evaluate the benefits of exploiting the CFG of the program; then, we evaluate the effect of using interpolants for computing sets of inductive clauses (TREE-IC3+ITP) wrt. “plain” TREE-IC3; third, we evaluate the impact of the fine-grained strengthening that is possible when using a conjunctively-partitioned representation for abstract states, by comparing TREE-IC3+ITP with TREE-IC3+ITP-MONO. The rest of the plots show instead the comparison of our best configuration, TREE-IC3+ITP, with alternative algorithms and implementations. A summary of the performance results for all the algorithms/configurations is reported in the table, showing the number of instances successfully checked within the timeout, and the total execution time for the solved instances.

From the plots and the table of Figure 4, we can draw the following conclusions:

- All the techniques proposed in this paper lead to significant improvements to IC3, with TREE-IC3 solving 17 more instances than IC3 (and being up to two orders of magnitude faster), and TREE-IC3+ITP solving 11 more instances than TREE-IC3;
- On relatively-easy problems, the IC3-based algorithms are generally more expensive than the alternative techniques; in particular, the tools based on predicate abstraction (KRATOS and CPACHECKER) perform very well in terms of execution time. However, on harder benchmarks TREE-IC3+ITP seems to be more robust than the competitors. This is particularly evident for TREE-ITP and CPACHECKER, which run out of memory in 25 and 34 cases respectively,¹¹ whereas this never happens with KRATOS and TREE-IC3+ITP.
- The ability to perform clause-by-clause strengthening is very important for the performance of TREE-IC3+ITP: when using a “monolithic” approach, TREE-IC3+ITP (TREE-IC3+ITP-MONO) is not only almost always slower, but it also solves 13 instances less. However, we notice that even without it, TREE-IC3+ITP-MONO behaves better than TREE-ITP, and in particular it is significantly more robust in terms of memory consumption.

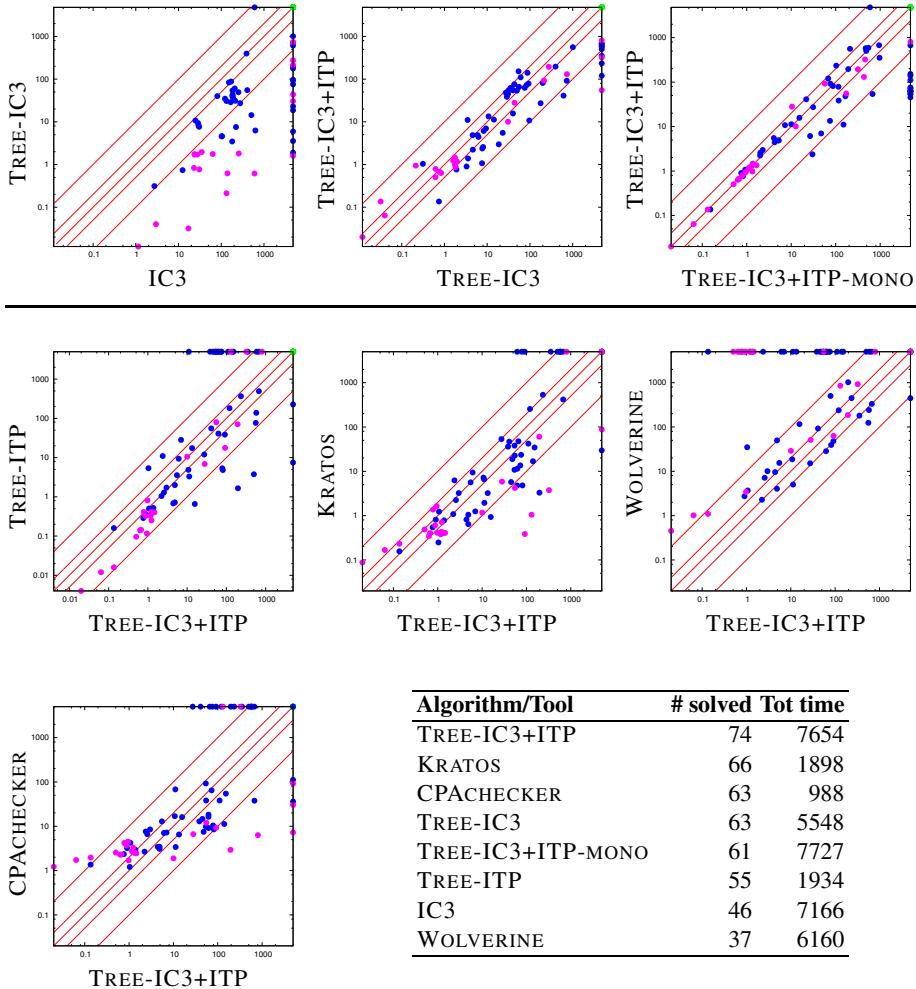
7 Related Work

Besides all the IC3-related approaches in the hardware domain [6][7][12], as already stated in §5 the work that is most closely-related to ours is the “lazy abstraction with

⁹ See <http://sv-comp.sosy-lab.org/results/index.php>

¹⁰ It would have been interesting to include also the IMPACT tool of [18] in the comparison; however, the tool is no longer available.

¹¹ Notice that with CPACHECKER this happens even when increasing the memory limit to 4GB.

**Fig. 4.** Experimental results

“interpolants” technique of McMillan [18]. Both TREE-IC3 and TREE-IC3+ITP, in fact, can be seen as instances of the “lazy abstraction with interpolants” algorithm, in which however interpolants are computed using the IC3 algorithm and approximate preimage computations, rather than proofs of unsatisfiability produced by the SMT solver. This in turn leads to interpolants that can be easily partitioned conjunctively, which allows to significantly improve their usefulness in pruning the number of abstract paths that need to be explored (see §5 and §6).

Another approach based on interpolation and explicit exploration of CFGs is described in [19]. In this approach, the search in the CFG is guided by symbolic execution; moreover, a learning procedure inspired by DPLL-based SAT solvers is applied in order to generate new annotations that prevent the exploration of already-visited portions of the search space. Such annotations are obtained from interpolants, generated from

proofs. In principle, it should be possible to apply IC3-based ideas similar to those that we have presented also in that context.

Some analogies between the present work and the “DASH” approach described in [2][13], which analyzes programs with a combination of testing and verification, can be seen in the use of approximate preimages and of highly-incremental SMT queries. However, in another sense DASH is somewhat orthogonal to IC3-based techniques, in that the latter could be used as a verification engine for the former. (In fact, although in [2] an approach based on weakest preconditions is used, interpolation is suggested as a potential alternative.)

Finally, the use of a clause-based representation for abstract states bears some similarities with the work in [15]. However, the exploration of the CFG and the whole verification approach is very different, and the two approaches can be considered largely orthogonal. In [15], the CFG is treated as a Boolean formula, whose satisfying assignments, enumerated by a SAT solver, correspond to path programs that are checked with different verification oracles. The invariants computed by such oracles are then used to construct blocking clauses that prevent re-exploration of already-covered parts of the program. Both the fully-symbolic and the tree-based versions of IC3 that we have presented could be used as oracles for checking the path programs and generating invariants for the blocking clauses.

8 Conclusions and Future Work

We have presented an investigation on the application of IC3 to the case of software. We propose three variants: the first one, generalizing IC3 to the case of SMT, provides for the analysis of fully symbolically represented software; the second one, TREE-IC3, relies on an explicit treatment of the CFG; the third one is a hybrid approach based on the use of interpolants to improve TREE-IC3.

IC3 is a radically new verification paradigm, and has a great potential for future developments in various directions. First, we intend to investigate further IC3 in the setting of SMT, devising effective procedures for APPROX-PREIMAGE in other relevant theories, and adding low-level optimization techniques, similarly to the highly tuned techniques used in the Boolean case. Second, we intend to investigate the extraction of CFG’s from hardware designs, in the same spirit as the transition-by-transition approach [21], and to apply IC3 directly to descriptions in high-level languages such as Verilog or VHDL. Finally, we intend to extend IC3 to richer theories such as bit vectors and arrays, and to the case of networks of hybrid systems [10].

Acknowledgements. We would like to thank Ken McMillan and the anonymous referees for their very helpful comments on early versions of the paper.

References

1. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, vol. 185, pp. 825–885. IOS Press (2009)
2. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proc. of ISSTA, pp. 3–14. ACM (2008)

3. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via Large-Block Encoding. In: Proc. of FMCAD, pp. 25–32. IEEE (2009)
4. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
5. Bradley, A., Somenzi, F., Hassan, Z., Zhang, Y.: An incremental approach to model checking progress properties. In: Proc. of FMCAD (2011)
6. 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)
7. Chokler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: Proc. of FMCAD (2011)
8. Cimatti, A., Griggio, A., Micheli, A., Narasamdy, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)
9. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Log. 12(1), 7 (2010)
10. Cimatti, A., Mover, S., Tonetta, S.: Efficient Scenario Verification for Hybrid Automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 317–332. Springer, Heidelberg (2011)
11. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
12. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property-directed reachability. In: Proc. of FMCAD (2011)
13. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Proc. of POPL, pp. 43–56. ACM (2010)
14. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8 (2012)
15. Harris, W.R., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Proc. of POPL, pp. 71–82. ACM (2010)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. of POPL, pp. 58–70 (2002)
17. Kroening, D., Weissenbacher, G.: Interpolation-Based Software Verification with WOLVERINE. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)
18. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
19. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
20. Monniaux, D.: A Quantifier Elimination Algorithm for Linear Real Arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 243–257. Springer, Heidelberg (2008)
21. Nguyen, M.D., Stoffel, D., Wedler, M., Kunz, W.: Transition-by-transition FSM traversal for reachability analysis in bounded model checking. In: Proc. of ICCAD. IEEE (2005)
22. Roorda, J.-W., Claessen, K.: SAT-Based Assistance in Abstraction Refinement for Symbolic Trajectory Evaluation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 175–189. Springer, Heidelberg (2006)
23. Somenzi, F., Bradley, A.: IC3: Where Monolithic and Incremental Meet. In: Proc. of FMCAD (2011)

Delayed Continuous-Time Markov Chains for Genetic Regulatory Circuits*

Călin C. Guet, Ashutosh Gupta, Thomas A. Henzinger,
Maria Mateescu, and Ali Sezgin

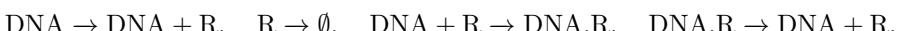
IST Austria, Klosterneuburg, Austria

Abstract. Continuous-time Markov chains (CTMC) with their rich theory and efficient simulation algorithms have been successfully used in modeling stochastic processes in diverse areas such as computer science, physics, and biology. However, systems that comprise non-instantaneous events cannot be accurately and efficiently modeled with CTMCs. In this paper we define delayed CTMCs, an extension of CTMCs that allows for the specification of a lower bound on the time interval between an event’s initiation and its completion, and we propose an algorithm for the computation of their behavior. Our algorithm effectively decomposes the computation into two stages: a pure CTMC governs event initiations while a deterministic process guarantees lower bounds on event completion times. Furthermore, from the nature of delayed CTMCs, we obtain a parallelized version of our algorithm. We use our formalism to model genetic regulatory circuits (biological systems where delayed events are common) and report on the results of our numerical algorithm as run on a cluster. We compare performance and accuracy of our results with results obtained by using pure CTMCs.

1 Introduction

Due to the Brownian motion of molecules inside cells, biological systems are inherently stochastic. The stochastic effects are negligible when all species are present in large numbers, but can be significant when some of the species are present only in low numbers. In particular, when modeling genetic regulatory circuits (GRCs), where different molecules (such as DNA) are present in low numbers, one needs to take stochasticity into account. Indeed, systems biology has been shifting its focus from deterministic models that capture the mean behavior of GRCs to stochastic models that capture their stochastic behavior [9].

One of the most general modes of gene regulation is self-regulation in the form of a *negative feedback* loop [19], where a DNA molecule encodes a repressor protein R with which it interacts according to the following reactions:



* This work was supported by the ERC Advanced Investigator grant on Quantitative Reactive Modeling (QUAREM) and by the Swiss National Science Foundation.

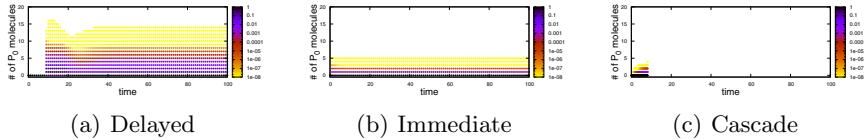


Fig. 1. Probability distribution of the levels of repressor protein over time in three different models of the negative feedback loop. (a) an overshoot is observed in the delayed CTMC model; (b) steady state is reached rapidly in the immediate CTMC; (c) the model is intractable in the cascade CTMC and thus we present only the distribution up to time 7.2s.

that correspond to the production, degradation, binding, and unbinding of protein. Due to biological aspects, the production of proteins behaves as a process with high latency and high throughput. Therefore, if at time $t = 0$ the system contains a single DNA molecule, the production of a large number of proteins is initiated because, before the completion of any of the protein production processes, nothing inhibits the production of proteins (Fig. 1(a)). Consequently, the observed behavior of the system is crucially dependent on the presence of delays, a well known phenomenon in biological systems [59][11].

A classic modeling formalism for GRCs is offered by continuous-time Markov chains (CTMCs) as proposed by Gillespie [7]. Under the assumptions that the solution is well-stirred, at constant temperature and volume, and that reactions happen instantaneously, the CTMC model considers (i) that the state of the system is given by a population vector whose elements represent the copy number for each molecule type, and (ii) that reactions are triggered at times that follow an exponential distribution, with a rate that depends on the quantum mechanical properties of molecules and on the likelihood of their collision. A popular technique for analyzing this type of models is *probability propagation* [12], which computes the transient probabilities of the Markov chain over time by pushing probability distribution through the state space of the model.

In the classical CTMC model of the negative feedback system each state of the model has three variables that denote the number of DNA (0 or 1), DNA.R (0 or 1) and R (a natural number) molecules. Each state has up to four successors, one for each of the enabled reactions (production, degradation, binding and unbinding of repressor protein). Because reactions happen instantaneously, without any latency, with some strictly positive probability proteins are available at any $t > 0$, and thus can inhibit further production of proteins immediately. The overshooting behavior that is normally present in the negative feedback loop is thus not observed in this *immediate CTMC* model (Fig. 1(b)). In order to overcome this problem, a pure CTMC model needs to use additional auxiliary variables that encode the age of a molecule, and thus produce a delay-like behavior. This increase in the number of variables leads however to a state space explosion that makes even simple models intractable (Fig. 1(c)). We call such a CTMC, extended with auxiliary variables, a *cascade CTMC*.

In this paper, we introduce *delayed CTMCs*, a formalism that we argue to be more natural and more efficient than pure CTMCs when modeling systems that comprise non-instantaneous reactions. In a delayed CTMC with delay Δ , each species x has an associated age $\alpha(x)$ that specifies that $(\alpha(x) - 1) \cdot \Delta$ time units must pass between the moment when a reaction that produces x is triggered and the moment when x is available as a reactant of a new reaction.

Natural. Delayed CTMCs naturally express non-instantaneous reactions because both the throughput and the latency of a reaction have direct correspondents in the rate of the reaction and the delay of the reaction, respectively. Even though one can try to model both the latency and throughput of reactions in a pure CTMC by adding auxiliary reactions, the determination of the number and parameters of such reactions involve the manipulation of complex functions. Furthermore, one cannot simply approximate the computation of these parameters because it is not clear how such approximations affect the qualitative behavior of the model.

Efficient. Delayed CTMC have two important performance advantages with respect to cascade CTMCs. First, by decoupling the waiting times of molecules in the process of being produced from the dynamics of the currently available molecules, we reduce the size of the state space on which to run a pure CTMC computation. Second, since no probability can flow between states with different number of waiting molecules (molecules that are not yet “mature”), our probability transition matrix accepts a simple partitioning, which is efficiently parallelizable.

The algorithm that we propose for the computation of the probability propagation of the delayed CTMC consists of alternating rounds of pure CTMC computation and aging steps. Each pure CTMC computation can be parallelized due to the absence of interactions between subspaces of the model, and thus we are able to solve delayed CTMC models that have large state spaces. For example, we solve the negative feedback example for parameters that generate a state space of up to 112 million states. The result of these experiments (see Figure 1) show that the delayed CTMC model of the negative feedback system indeed matches the experimental evidence of an initial overshoot in the production of protein 9, while the pure CTMC models do not.

Reaction delays have already been embedded in stochastic simulation tools for GRCs [18], but we are not aware of any work that embeds such delays in a probability propagation algorithm. The probability propagation problem relates to stochastic simulations as verification relates to testing. Due to the advantages of probability propagation algorithms over simulation based approaches (when computing the transient probabilities of a system) [3], it is valuable to provide such algorithms. As probability propagation of stochastic systems is analogous to solving reachability problems in the non-stochastic setting, we use techniques such as discretization of a continuous domain and on-the-fly state space exploration, which are standard techniques for verification problems.

Related Work. There has been a wide interest in defining formalisms for genetic regulatory circuits [27][15][17]. In particular, using delays within stochastic simulations has recently drawn interest (see [18] for a review of these models). Delayed CTMCs differ from these models in that they discretize the delay time as opposed to having continuous delays.

There has also been much work on the transient analysis of pure CTMCs coming especially from the field of probabilistic verification [8][10][13][14], but none of these methods consider reaction delays.

Recent efforts [1][20] have parallelized the transient analysis of CTMCs by applying parallel algorithms for sparse matrix multiplication. Since the data dependencies flow across the entire state space, they have achieved limited speed-ups. Our work is orthogonal to these approaches. Due to the nature of delayed CTMCs, the probability transition matrix of the model is amenable to being partitioned into disconnected sub-matrices, and thus we obtain a highly parallelizable algorithm. This can lead to large speed-ups (hundreds of times for our examples). The techniques presented in these related works can be used for further parallelization of our algorithm by using them in the computation of each disconnected part of the matrix.

Delayed CTMCs are a subclass of generalized Markov processes (GMPs). Our extension of CTMCs is limited as compared to GMPs, in that our single extension is in capturing the property of latent reactions. However, we are able to find efficient probability propagation algorithms for delayed CTMCs.

2 Delayed CTMC

For a set A , let $f|_A$ denote the function obtained by restricting the domain of f to $A \cap \text{Dom}(f)$. $\text{Pow}(A)$ denotes the powerset of A .

Continuous-Time Markov Chain (CTMC). A probability distribution ρ over a countable set A is a mapping from A to $[0, 1]$ such that $\sum_{a \in A} \rho(a) = 1$. The set of all probability distributions over A is denoted by \mathcal{P}^A . The *support* F_ρ of a probability distribution ρ over A is the subset of A containing exactly those elements of A mapped to a non-zero value. Formally, $F_\rho = \{a \in A \mid \rho(a) \neq 0\}$.

A CTMC M is a tuple (S, Λ) , where S is the set of *states*, and $\Lambda : S \times S \mapsto \mathbb{R}$ is the *transition rate matrix*. We require that for all $s, s' \in S$, $\Lambda(s, s') \geq 0$ iff $s \neq s'$, and $\sum_{s' \in S} \Lambda(s, s') = 0$.

The *behavior* of $M = (S, \Lambda)$ is a mapping \mathbf{p}_M from \mathbb{R} to a probability distribution over S satisfying the *Kolmogorov differential equation*

$$\frac{d}{dt} \mathbf{p}_M(t) = \mathbf{p}_M(t) \cdot \Lambda$$

If the value of $\mathbf{p}_M(0)$ is known, the above differential equation has the unique solution

$$\mathbf{p}_M(t) = \mathbf{p}_M(0) \cdot e^{\Lambda t}$$

In the case where $|S| < \infty$, the series expansion for e^{At} yields $\sum_{i=0}^{\infty} (At)^i / i!$ for which analytic solutions can be derived only for special cases. In general, finding the probability distribution \mathbf{p}_M as a symbolic function of time (t) is not possible.

We will let $M(\rho, t)$, the behavior of M initiated at ρ , denote the value of $\mathbf{p}_M(t)$ with $\mathbf{p}_M(0) = \rho$.

Aging Boundary, Configurations. An *aging boundary* is a pair (X, α) , where X is a finite set of *variables*, $\alpha : X \mapsto \mathbb{N}$ is an *age* function. The *expansion* of the aging boundary (X, α) , is the set $[X, \alpha] = \{(x, a) \mid x \in X, 0 \leq a \leq \alpha(x)\}$, the elements of which are called *aged variables*. For an expansion $[X, \alpha]$, we define the sets of *immediate*, *new* and *waiting* variables as $[X, \alpha]^i = \cup_{x \in X} \{(x, 0)\}$, $[X, \alpha]^n = \cup_{x \in X} \{(x, \alpha(x))\}$, and $[X, \alpha]^w = \{(x, a) \mid x \in X, 0 < a < \alpha(x)\}$, respectively.

A *configuration* c over an expansion $[X, \alpha]$ is a total function from $[X, \alpha]$ to \mathbb{N} . We will also write $((x, a), n) \in c$ whenever $c(x, a) = n$. A *sub-configuration* c_F of c relative to $F \subseteq [X, \alpha]$ is the restriction of c to F ; formally, $c_F(x)$ is defined to be equal to $c(x)$ iff $x \in F$. For any $F \subseteq [X, \alpha]$, \mathcal{C}^F denotes the set of all sub-configurations over F . For any configuration $c \in \mathcal{C}^{[X, \alpha]}$, let c_i , c_n and c_w denote the sub-configurations of c relative to $[X, \alpha]^i$, $[X, \alpha]^n$ and $[X, \alpha]^w$, respectively. Intuitively, in the context of gene expression, an aged variable will represent a molecule with a time stamp denoting the delay until the molecule is produced, and configurations will represent a collection of molecules with time stamps. In what follows, we will be exclusively working on CTMCs having configurations as states, and thus will use the two terms, states and configurations, interchangeably.

Delayed CTMCs. Let (X, α) be an aging boundary. A CTMC $M = (\mathcal{C}^{[X, \alpha]}, \Lambda)$ is α -safe, if $\Lambda(c, c') \neq 0$ implies that $c'_w = c_w$ and also, $c'(x, \alpha(x)) < c(x, \alpha(x))$ implies $\alpha(x) = 0$. Intuitively, α -safe means that the waiting variables cannot change and only the value of immediate variables can decrease.

Definition 1 (Delayed CTMC). A delayed CTMC D is a tuple $(X, \alpha, \Lambda, \Delta)$, where (X, α) is an aging boundary, $M_D = (\mathcal{C}^{[X, \alpha]}, \Lambda)$ is α -safe, and $\Delta \in \mathbb{R}^+$ is the delay.

A *behavior* of a delayed CTMC $D = (X, \alpha, \Lambda, \Delta)$ is a finite sequence $\rho_0 \rho_1 \rho_2 \dots \rho_N$ of probability distributions over $\mathcal{C}^{[X, \alpha]}$ that satisfies

$$\rho_{i+1} = \text{Tick}(M_D(\rho_i, \Delta)) \quad \text{for } 0 \leq i < N \quad (1)$$

The definition of *Tick* is given as

$$\text{Tick}(\rho)(c') \stackrel{\text{def}}{=} \sum_{c^{+1}=c'} \rho(c), \quad c^{+1}(x, a) = \begin{cases} c(x, a+1) & , \text{if } 0 < a < \alpha(x) \\ c(x, a+1) + c(x, a) & , \text{if } a = 0 \\ 0 & , \text{if } a = \alpha(x) \end{cases}$$

Intuitively, a *behavior* of a delayed CTMC D is an alternating sequence of running the CTMC M_D for Δ units of time, *deterministically* decrementing the age

of each variable in every configuration (i.e. propagating probability from c to c^{+1}), and computing the new probability distribution (Tick).

A *continuous behavior* of $D = (X, \alpha, \Lambda, \Delta)$ is given as

$$\mathbf{b}_D(t) \stackrel{\text{def}}{=} M_D(\rho_q, z)$$

where $t = q \cdot \Delta + z$ for some $0 \leq z < \Delta$, and $\rho_0 \dots \rho_q$ is a behavior of D .

3 Genetic Regulatory Circuits

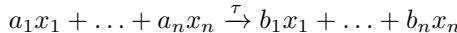
In this section, we will first give a simple formalism for defining genetic regulatory circuits. We will then provide three different semantics for genetic regulatory circuits using delayed CTMCs.

3.1 Specifying Genetic Regulatory Circuits

A genetic regulatory circuit (GRC) G is a tuple (X, α, R) , where

- (X, α) is an aging boundary,
- R is a set of *reactions*.

Each reaction $r \in R$ is a tuple (i_r, τ, o_r) , where i_r and o_r , the *reactant* and *production* list, respectively, are mappings from X to \mathbb{N} , and τ , the *reaction rate*, is a positive real-valued number. For reactions, we will use the more familiar notation



where $a_j = i_r(j)$, and $b_j = o_r(j)$. Intuitively, each reaction represents a chemical reaction, and variables of X represent the molecular species that take part in at least one reaction of the system. Each reaction defines the necessary number of each molecular species that enables a chemical reaction, the base rate of the reaction, and the number of produced molecular species as a result of this chemical reaction.

Overall, a reaction can be seen as a *difference vector* $\mathbf{r} = [r_1 \ r_2 \ \dots \ r_n]$, where $r_i = o_r(x_i) - i_r(x_i)$ is the net change in the number of molecule x_i . We will assume that the difference vector of each r is unique. In writing down a reaction, we will leave out the molecular species that are mapped to 0.

3.2 Dynamics in Terms of Delayed CTMCs

In this section, we will give the semantics of a GRC $G = (X, \alpha, R)$ in terms of a delayed CTMC. A computation framework for G is parameterized over delay values. For the following, we fix a delay value, Δ .

A reaction $r \in R$ is *enabled* in a configuration c , if for all $x_i \in X$, $c(x_i, 0) \geq i_r(x_i)$ holds. In other words, reaction r is enabled in c if the number of reactants that r requires is at most as high as the number of immediately available reactants in c . Let $En(c) \subseteq R$ denote the set of reactions enabled in c .

For configurations c, c' , and reaction $r \in R$, we say that c can go to c' by firing r , written $c \xrightarrow{r} c'$, if $r \in En(c)$, and there exists a configuration \hat{c} such that

- c and \hat{c} are the same except for all $x_i \in X$, $\hat{c}(x_i, 0) = c(x_i, 0) - i_r(x_i)$, and
- c' and \hat{c} are the same except for all $x_i \in X$, $c'(x_i, \alpha(x_i)) = \hat{c}(x_i, \alpha(x_i)) + o_r(x_i)$.

Informally, to move from configuration c to c' via reaction r , c must have at least as many *immediate* molecules as required by the reactant list of r , and c' is obtained by removing all immediate molecules consumed by r and adding all the *new* molecules produced by r .

Delayed Semantics. For $G = (X, \alpha, R)$, we define the delayed CTMC $D_G = (X, \alpha, \Lambda, \Delta)$. We only need to give the definition of Λ .

G induces the transition rate matrix Λ defined as $\Lambda(c, c') = \text{Fire}(c, r)$ only when $c \xrightarrow{r} c'$ holds. $\text{Fire}(c, r)$ is given as

$$\text{Fire}(c, r) = \prod_{x_i \in X} \tau_r \binom{c(x_i, 0)}{i_r(x_i)}$$

where $\binom{n}{r} = n!/(n - r)!$ represents the choose operator. $\Lambda(c, c')$ is well-defined because there can be at most one reaction that can satisfy $c \xrightarrow{r} c'$ since we assumed that the difference vector of each reaction is unique. Observe also that no changes to waiting variables can happen in any transition with non-zero rate, and only the number of immediate variables can decrease. Hence, M_{D_G} as defined is α -safe.

Immediate Semantics. Given a GRC $G = (X, \alpha, R)$, we define the *immediate* version of G , written G^\downarrow as the GRC (X, α', R) , where $\alpha'(x) = 0$, for all $x \in X$. Intuitively, G^\downarrow ignores all the delays, and treats the reactions as instantaneously generating their products. Note that, a delayed CTMC with an age function assigning 0 to all the variables is a pure CTMC. The immediate semantics for G are given by the behavior of the (delayed) CTMC constructed for G^\downarrow .

Cascade Semantics. Given a GRC $G = (X, \alpha, R)$, we define the *cascade* version of G , written as G^* , as the GRC (X', α', R') , where $X' = [X, \alpha]$, $\alpha'(x) = 0$, for all $x \in X'$, and $R' = R_{\text{trig}} \cup R_{\text{age}}$, where

- R_{trig} is the set of reactions of R re-written in a way that all the reactants have age 0, and all the products have their maximum age. Formally, for each reaction $r = \sum_i a_i x_i \xrightarrow{\tau} \sum_i b_i x_i \in R$, we define $\tilde{r} = \sum_i a_i(x_i, 0) \xrightarrow{\tau} \sum_i b_i(x_i, \alpha(x_i))$ and let $R_{\text{trig}} = \{\tilde{r} \mid r \in R\}$.
- R_{age} is the representation of delays in terms of a sequence of fictitious events intended to count down the necessary number of stages. Formally, $R_{\text{age}} = \{(x, a) \xrightarrow{\Delta^{-1}} (x, a - 1) \mid a > 0, (x, a) \in X'\}$.

Cascade semantics for G are given by the behavior of the (delayed) CTMC constructed for G^* .

Remark. A protein generated by a GRC has two dynamic properties: the rate of its production and the delay with which it is produced, both of which are

intuitively captured by delayed semantics. The immediate semantics keeps the rate intact at the expense of the production delay which is reduced to 0. On the other hand, the cascade semantics approximates the delay, by a chain of events with average delay Δ , while increasing the overall variance of the production time. In Section 4.1 we show that a close approximation of the mean and the variance of a delayed CTMC by cascade semantics causes a blow-up of the state space of the former by $O(\alpha(x)^2 \tau_r)$, where r is a reaction producing protein x .

4 Comparison of Different Semantics for GRCs

In this section, we will do a comparative analysis of the delayed CTMC model. First, we will derive the probability distribution expression for the three different semantics we have given in the previous section. We will show that delayed CTMCs are more succinct than cascade CTMCs. We will then give two examples which demonstrate the qualitative differences between the immediate, cascade and delayed semantics for the same GRC.

4.1 Probability Distributions for Delayed Reactions

Let $G = (X, \alpha, R)$ be a GRC, and let $x \in X$ be such that $\alpha(x) = k$, for some $k > 0$. We would like to analyze the probability distribution of the time of producing x due to a reaction $r \in R$ with $\alpha_r(x) > 0$ in the three different semantics we have given in the previous section. We use $Pr(t_p(x) \leq T)$ to denote the probability of producing x at most T time units after the reaction r took place.

For immediate semantics, the cumulative distribution is simply a step function, switching from 0 to 1 at time 0 since the initiation and completion times of a reaction in immediate semantics are equal. In other words, we have $Pr(t_p(x) \leq T) = 1$, for all $T \geq 0$.

For cascade semantics, we have k intermediate reactions, each with an identical exponential distribution. This means that the probability density function for producing x at exactly t units of time is given by the $(k - 1)$ convolution of the individual probability density functions. This is known as the Erlang distribution, and has the closed form $f_{k,\Delta}(t) = \frac{t^{k-1}}{(k-1)! \Delta^k} e^{-t/\Delta}$. The mean and the variance of this distribution are given as $k\Delta$ and $k\Delta^2$, respectively. This implies that as k increases, both the mean and the variance of the distribution increase, a fact which we show has an important consequence in terms of model size.

For delayed semantics, we know that for x at time $n\Delta$ to have, for the first time, age 0 which makes it immediate (and produced), the reaction r producing x must have occurred during the (half-open) interval $((n-k)\Delta, (n-k+1)\Delta]$. Since this means that the production time of x cannot be greater than $k\Delta$ and cannot be less than $(k-1)\Delta$, the probability density function of the production time of x due to r is non-zero only in the interval $[(k-1)\Delta, k\Delta]$. Let us denote this interval with $p^+(x, r)$. Let δ range over the real numbers in the interval $p^+(x, r)$. Then, the probability of x being produced by r in $(k-1)\Delta + \delta$ units of time

is equal to the probability of r taking place at $\Delta - \delta$ units of time given that r takes place in the interval $(0, \Delta]$. As the calculation of the transition rate matrix Λ has shown, the probability of reaction r firing depends on configurations; the base rate τ_r defines a lower bound on the actual rate of r . Since the lower the actual rate the higher the variation is, we are going to compute the distribution for the base rate. Then, the probability expression for $p^+(x, r)$ becomes

$$\Pr(t_p(x) \leq (k-1)\Delta + \delta) = 1 - \frac{1 - e^{-\tau_r(\Delta-\delta)}}{1 - e^{-\tau_r\Delta}}, \quad \delta \in [0, \Delta]$$

This expression shows that with increasing values of reaction rate τ_r , the probability of the production of x taking time close to $\alpha(x)$ also increases. This is expected since as the rate of the reaction r producing x gets higher, the probability of r taking place close to the beginning of the interval in which it is known to happen also gets higher. In other words, it is possible to generate a probability distribution for the production time of x such that

$$\Pr(\alpha(x) - \delta \leq t_p(x) \leq \alpha(x)) = 1 - \varepsilon$$

for arbitrary δ and ε , which we consider further in the rest of this subsection.

Quasi-periodicity of Delayed CTMCs. Previously, we have given three alternative semantics for GRCs. In giving cascade semantics, the intuition was to replace each *deterministic* aging step of the delayed CTMC with an intermediate fictitious aging reaction. For each intermediate reaction, we have chosen the rates to be the inverse of Δ so that the collective cascading behavior has a mean delay equal to the age of the produced element. As we shall see in the next section, this conversion leads to different qualitative behaviors.

We will now compare delayed CTMC to what we call cascade CTMCs, a generalization of the cascade semantics, and show that preserving a property called quasi-periodicity requires a blow-up in the state space while converting a delayed CTMC into a cascade CTMC.

Let \mathbf{b} be a continuous behavior of a delayed CTMC D . Recall that \mathbf{b} is a mapping from \mathbb{R} , representing time, to a probability distribution over some $\mathcal{C}^{[X, \alpha]}$. We will call \mathbf{b} *quasi-periodic* with $(\varepsilon, \delta, p, l)$ at configuration c if for all $t \leq l$, $\mathbf{b}(t)(c) \geq 1 - \varepsilon$ implies that there exists a number $k \in \mathbb{N}$ such that $t \in [kp, kp + \delta]$, and if $t \notin [kp, kp + \delta]$ for any $k < l$, then $\mathbf{b}(t)(c) \leq \varepsilon$. Intuitively, if the behavior \mathbf{b} is quasi-periodic with $(\varepsilon, \delta, p, l)$ at c , then the probability of visiting c is almost 1 (ε is the error margin) only at multiples of the period p within δ time units. In all other times, the probability of being in c is almost 0 (less than ε). The parameter l gives the period of valid behavior; nothing is required of \mathbf{b} for times exceeding l . Typically, we will want to maximize l , the length of the behavior displaying the desired behavior, and minimize ε and δ , the uncertainty in periodic behavior. Because periodicity is crucial during biological processes such as embryo development [16] or circadian clocks [5], quasi-periodicity defines an important subclass of behaviors.

For a set S , totally ordered by \prec , and elements $s, s' \in S$, let $s' = s + 1$ hold only when s' is the least element greater than s in S according to \prec . Let s_{min}

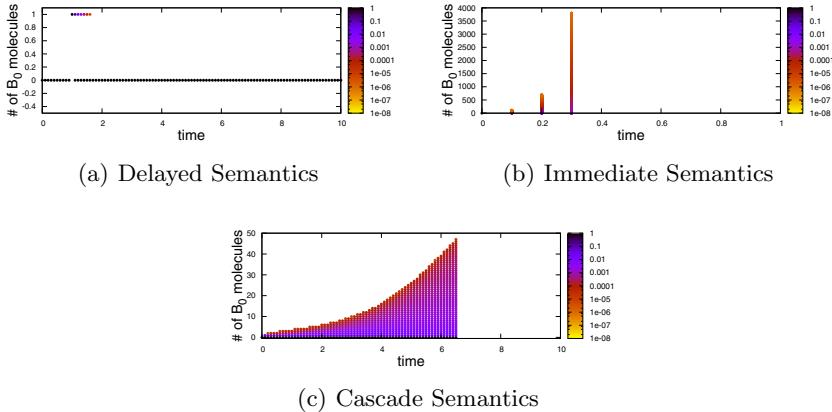


Fig. 2. Behaviors of the GRC ConvDiv. Due to intractability of the immediate and cascade models, (b) depicts the probability distribution up to time 0.3s and (c) represents the probability distribution up to time 6s.

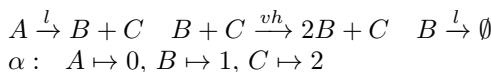
denote the minimal element in the totally ordered set S . If S is finite, then for the maximal element s_{max} of S , we let $s_{min} = s_{max} + 1$.

A CTMC $M = (S, \Lambda)$ is called a *cascade CTMC* if S is totally ordered, and $\Lambda(s, s') > 0$ iff $s' = s + 1$. A cascade CTMC is λ -homogeneous if $\Lambda(s, s') > 0$ iff $\Lambda(s, s') = \lambda$ or $s = s_{min}$. In other words, a cascade CTMC is λ -homogeneous if all the state transitions have the same rate λ with the possible exception of the transition out of the minimal state.

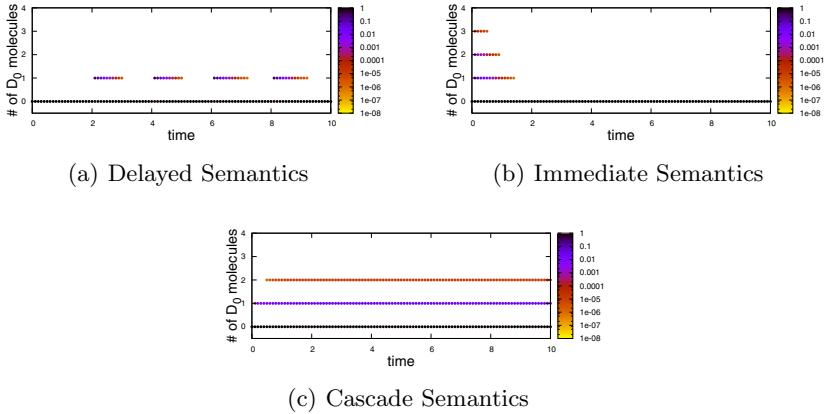
Theorem 1. *There exists a class of delayed CTMCs M_i that are quasi-periodic such that there is no corresponding class of λ -homogeneous cascade CTMCs M'_i with $|M'_i| = O(|M_i|)$.*

4.2 Examples Demonstrating Qualitatively Different Behavior

The first example GRC, ConvDiv, is given as

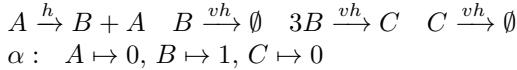


The symbols l , h , vh , represent low, high and very high rates, respectively. At $t = 0$ the model contains a single molecule, of type A . After the first reaction produces one B and one C , observe that the number of B molecules will increase only if the second reaction fires, which requires for both B and C to be present in the system. In immediate semantics, the expected behavior is divergence: the number of B molecules should increase without bound. However, when the delay values are taken into account, we see that B and C molecules with different ages are unlikely to be present in the system simultaneously. Thus, a stable behavior

**Fig. 3.** The behaviors of the GRC **Periodic**

should be observed for delayed semantics. Since cascade semantics still allow for non-zero probability of producing B and C , albeit at a lower probability, divergence should also be observed for cascade semantics. The computed behaviors given in Figure 2 are in accordance with our explanations.

The second example GRC, **Periodic**, is given as



We observe that the production rate of B from the first reaction is slower than the degradation rate of B , which means that in immediate semantics, it is very unlikely to have 3 B 's at any time, which in turn implies that C is not likely to be produced at all. However, in delayed semantics, A will keep producing B during Δ time units, which are likely to be more than 3 in the beginning of the next step. This increases the probability of producing C considerably. In fact, C must be exhibiting quasi-periodic behavior. The computed behaviors are given in Figure 3. As expected from the arguments of the previous section, cascade semantics, even though does not stabilize at $C = 0$ like the immediate semantics, still can not exhibit a discernible separation between the times where $C = 0$ and the times where $C > 0$.

Remark. The examples of this section illustrate the impact of incorporating delay into models. As for representing a given biological system in a GRC, some of the encoding issues pertain to a natural extension of the syntax. For instance, having different production delays for the same molecular species in different reactions or allowing more than one reaction with the same difference vector are simple extensions to our formalism. Another issue is that the (delayed) molecular species are produced at exact multiples of Δ ; this can be modified by using additional reactions. For instance, a reaction $A \xrightarrow{k} B$ with $\alpha(B) = 1$ can be

<pre> function COMPUTEBEHAVIOR input $D = (X, \alpha, \Lambda, \Delta)$: delayed CTMC ρ_0 : initial probability distribution n : time count W : number of workers $C^1, \dots, C^W \subseteq \mathcal{C}^{[X, \alpha]}$ assume $C^1 \uplus \dots \uplus C^W = \mathcal{C}^{[X, \alpha]}$ $\forall c, c' \in \mathcal{C}^{[X, \alpha]} \exists i. (c_w = c'_w \Rightarrow \{c, c'\} \subseteq C^i)$ output ρ_1, \dots, ρ_n : probability distributions begin for each $i \in 1..W$ do $\rho_0^i := \rho_0 _{C^i}$ done LAUNCHANDWAIT(WORKER, W) for each $k \in 1..n$ do $\rho_k := \rho_k^1 \cup \dots \cup \rho_k^W$ done end </pre>	<pre> function WORKER input (local) i : worker index locals $k, j : \mathbb{N}$ ρ : probability distribution begin $k := 0$ while $k < n$ do $\rho := \text{RUNCTMC}(\rho_k^i, M_D, \Delta)$ for each $c \in \text{Dom}(\rho)$ do j such that $c^{+1} \in C^j$ atomic{ $\rho_{k+1}^j(c^{+1}) := \rho_{k+1}^j(c^{+1}) + \rho(c)$ } done SYNCWORKERS() $k := k + 1$ done TERMINATIONSIGNAL() end </pre>
--	---

Fig. 4. A parallel algorithm for computing the behavior of a delayed CTMC. The call of LAUNCHANDWAIT starts W processes each running the function WORKER and waits for the execution of TERMINATIONSIGNAL call in all worker processes. SYNCWORKERS synchronizes all worker processes. We assume that all variables are global (including the input parameters of COMPUTEBEHAVIOR) except for the explicitly declared locals.

replaced with two reactions $A \xrightarrow{k} ZB$ and $ZB \xrightarrow{k_z} B$ with $\alpha(ZB) = 1$, $\alpha(B) = 0$, and where ZB is a fictitious molecular species. Then, adjusting the value of k_z will define a distribution for the production of the molecule B .

5 Behavior Computation of Delayed CTMC

In this section we present an algorithm for computing the behavior of a delayed CTMC given an initial state of the model. Since this behavior cannot be computed analytically, we propagate the probability distribution over time using Equation (II).

The configuration space of the delayed CTMC can be divided into subspaces such that, between two consecutive tick instants, probability is not propagated from one subspace to another. Let c and c' be two configurations with different values of their waiting variables, i.e. $c_w \neq c'_w$. Due to the α -safe property of the delayed CTMC, there can be no propagation of probability from c to c' between two consecutive tick instants. Therefore, the behaviors corresponding to each subspace can be computed independently for this time period, which has length Δ . For this computation we can use any pure CTMC behavior computation algorithm. Furthermore, these independent computations can be executed in

```

function SPACEID( $s$ )
begin
   $i := 1, idx := 0$ 
  for each  $(x, a) \in Dom(c_w)$  do //  $(x, a)$  with larger  $a$  is chosen first
     $idx := idx + 4^i s_w((x, a))$ 
     $i := i + 1$ 
  done
  return  $(i\%W) + 1$ 
end

```

Fig. 5. In our implementation, SPACEID is used to divide the state space in partitions for the worker processes

parallel. In the case of pure CTMC, a similar parallelization is not possible because the behavior dependencies flow through the full configuration space.

In Figure 4 we illustrate our parallel algorithm COMPUTEBEHAVIOR that computes the behavior of a delayed CTMC $D = (X, \alpha, \Lambda, \Delta)$ starting from initial distribution ρ_0 . The algorithm computes the behavior of D until n time steps, i.e., ρ_1, \dots, ρ_n . COMPUTEBEHAVIOR uses W number of worker processes to compute the probability distributions. Each of the W works is assigned a subspace of $\mathcal{C}^{[X, \alpha]}$, as decided by the input partitions C^1, \dots, C^W . These partitions must ensure that if two configurations have equal values of waiting variables then both configurations are assigned to the same worker.

COMPUTEBEHAVIOR divides ρ_0 into the sub-distributions $\rho_0^1, \dots, \rho_0^W$ according to the input partitions. Then, it launches W number of workers who operate using these initial sub-distributions. Workers operate in synchronized rounds from 0 to $n-1$. At the k -th round, they compute the probability sub-distributions of the $k+1$ -th time step $\rho_{k+1}^1, \dots, \rho_{k+1}^W$. The i -th worker first runs a standard CTMC behavior computation algorithm RUNCTMC on ρ_k^i that propagates the probability distribution until Δ time and the final result of the propagation is stored in ρ . Then, the inner loop of the worker applies Tick operation on ρ . For each configuration c in $Dom(\rho)$, Tick adds $\rho(c)$ to the probability of c^{+1} in the appropriate sub-distribution decided by the configuration space partitions. Note that a configuration c may be the successor of many configurations, i.e., there may exist two configurations c_1 and c_2 such that $c_1^{+1} = c_2^{+1} = c$. and multiple workers may access $\rho_{k+1}^j(c)$, where j is such that $c \in C^j$. Therefore, we require this update operation to be atomic. After the Tick operation, workers move to next round synchronously. After all workers terminate their jobs, COMPUTEBEHAVIOR aggregates the sub-distributions into full distributions for each time step and produces the final result.

6 Implementation and Results

Implementation. We extended the SABRE-toolkit [4], a tool for probability propagation of pure CTMCs, to solve delayed CTMCs. We implemented COMPUTEBEHAVIOR as a multi-process system with an inter-process

Semantics	Example	Figure	Time Horizon	Run Time	Avg. Space	Qualitative Behavior
Delayed CTMC	ConvDiv	2(a)	10s	23s	695	converge
	Periodic	3(a)	10s	< 1s	13	periodic
	Feedback	1(a)	7.2s	23m	9×10^5	overshoot
	Feedback	1(a)	100s	7.65h \times 200*	3.2×10^7	overshoot
Cascade CTMC	ConvDiv	2(c)	6s	311m	119344	diverge
	Periodic	3(c)	10s	3s	351	uniform
	Feedback	1(c)	7.2s	21h	5.00×10^6	intractable
Immediate CTMC	ConvDiv	2(b)	0.3s	40m	4007	diverge
	Periodic	3(b)	10s	1s	26	decay
	Feedback	1(b)	100s	9s	96	fast stable

Fig. 6. Performance results for computing the behaviors corresponding to the examples presented earlier in this paper. We computed the behaviors until the time horizons within the run times. The avg. space column shows the configuration space with significant probabilities during the computation. In the **Feedback** example we use the following reaction rates: production = 1, binding = 1, unbinding=0.1, degradation=0.2. We assume that R remains latent 9 seconds after its production. We use multiple workers only for the **Feedback** example (*run time \times number of workers). The last column provides an intuitive description of the observed qualitative behavior.

communication implemented using MPI [6]. We use an implementation of the fast adaptive uniformization method [13] for RUNCTMC. Furthermore, we use function SPACEID, shown in the Figure 5, to define the partitions on the space: $C^i = \{c \in C^{[X, \alpha]} | \text{SPACEID}(c) = i\}$. In our examples, this policy leads to fairly balanced partitions of configurations among processes.

Experiments. In Figure 6 we present performance results for the behavior computation of the three discussed examples under the three semantics that we have introduced. We observe that delayed CTMCs offer an efficient modeling framework of interesting behaviors such as overshooting, convergence and periodicity.

We applied our implementation of COMPUTEBEHAVIOR on **Feedback** with delayed CTMC semantics using 200 workers. We were able to compute the behaviour until 100 seconds in 7.65 hours. COMPUTEBEHAVIOR with a single worker was able to compute the behaviour of **Feedback** until 7.2 seconds in 23 minutes, and for the same time horizon the behavior computation of **Feedback** under cascade semantics using sequential RUNCTMC took 21 hours to complete. Since the cascade CTMCs cannot be similarly parallelized, they suffer from a state space blowup and the negative impact on the performance cannot be avoided. In the case of immediate CTMC semantics, even if computing the behavior is relatively faster (except for **ConvDiv**, which has diverging behavior under the immediate CTMC semantics) the observed behavior does not correspond to the expectations of the model.

We also ran COMPUTEBEHAVIOR on **Feedback** for different values of Δ . Since the reaction rates in the real time remain the same, $\alpha(R)$ changes with varying Δ . In Figure 7(a), we plot expected values of R at different times for three values of $\alpha(R)$. We observe that with increasing precision, i.e. smaller Δ and higher $\alpha(R)$,

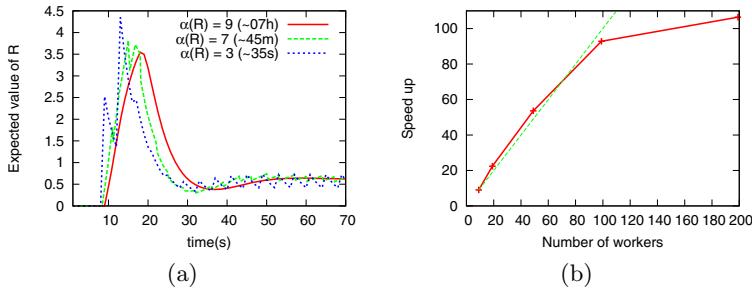


Fig. 7. (a) Expected numbers of molecules of protein R with varying $\alpha(R)$ in **Feedback**. We show the run time for computing each behavior in the legends. (b) Speed up vs. number of workers (negative feedback with $\alpha(R) = 5$).

the computed behaviors are converging to a limit behavior, but the running time of the computation increases rapidly, which is due to the the increase in number of aged variables causing an exponential blowup in configuration space. In Figure 7(b), we show the speed of computing the behaviour of **Feedback** with $\alpha(R) = 5$ for different number of workers. We observe that up to 100 workers the performance improves linearly, and there is no significant speedup after 100 workers. This is because each worker, when there are many of them, may not have significant computations per round, and communication and synchronization costs become the dominating factor.

7 Conclusion

Much like the introduction of time by time automata into a frame which was capable of representing ordering patterns without the ability to quantify these orderings more directly and accurately, we extended the widely used CTMC formalism by augmenting it with a time component in order to capture the behavior of biological systems containing reactions of relatively different durations, e.g. DNA transcription versus molecule bindings. We argue that our formalism achieves a more natural way to model such systems than CTMC (possibly extended with auxiliary reactions). We show that our approach also provides an efficient way of parallelizing the behaviour computation of the model.

As a continuation of this work, we are currently developing synthetic biology experiments meant to validate our predictions for the behavior of the **Periodic** example introduced in this paper.

References

1. Bosnacki, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel probabilistic model checking on general purpose graphics processors. *STTT* 13(1), 21–35 (2011)
2. Ciocchetta, F., Hillston, J.: Bio-pepa: A framework for the modelling and analysis of biological systems. *Theor. Comput. Sci.* 410(33–34), 3065–3084 (2009)

3. Didier, F., Henzinger, T.A., Mateescu, M., Wolf, V.: Approximation of Event Probabilities in Noisy Cellular Processes. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS(LNBI), vol. 5688, pp. 173–188. Springer, Heidelberg (2009)
4. Didier, F., Henzinger, T.A., Mateescu, M., Wolf, V.: Sabre: A tool for stochastic analysis of biochemical reaction networks. In: QEST, pp. 193–194 (2010)
5. Duong, H.A., Robles, M.S., Knutti, D., Weitz, C.J.: A Molecular Mechanism for Circadian Clock Negative Feedback. *Science* 332(6036), 1436–1439 (2011)
6. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (September 2004)
7. Gillespie, D.T.: A general method for numerically simulating the time evolution of coupled chemical reactions. *J. Comput. Phys.* 22, 403–434 (1976)
8. 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)
9. Joh, R.I., Weitz, J.S.: To lyse or not to lyse: Transient-mediated stochastic fate determination in cells infected by bacteriophages. *PLoS Comput. Biol.* 7(3), e1002006 (2011)
10. Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism 2.0: A tool for probabilistic model checking. In: QEST, pp. 322–323 (2004)
11. Maithreye, R., Sarkar, R.R., Parnaik, V.K., Sinha, S.: Delay-induced transient increase and heterogeneity in gene expression in negatively auto-regulated gene circuits. *PLoS ONE* 3(8), e2972 (2008)
12. Mateescu, M.: Propagation Models for Biochemical Reaction Networks. Phd thesis, EPFL, Switzerland (2011)
13. Mateescu, M., Wolf, V., Didier, F., Henzinger, T.A.: Fast adaptive uniformisation of the chemical master equation. *IET Systems Biology* 4(6), 441–452 (2010); 3rd q-bio Conference on Cellular Information Processing, St John Coll, Santa Fe, NM (August 05-09, 2009)
14. Munsky, B., Khammash, M.: The finite state projection algorithm for the solution of the chemical master equation. *J. Chem. Phys.* 124, 044144 (2006)
15. Myers, C.: Engineering genetic circuits. Chapman and Hall/CRC mathematical & computational biology series. CRC Press (2009)
16. Oswald, A., Oates, A.: Control of endogenous gene expression timing by introns. *Genome Biology* (3) (2011)
17. Regev, A., Silverman, W., Shapiro, E.: Representation and simulation of biochemical processes using the pi-calculus process algebra. In: Pacific Symposium on Biocomputing, pp. 459–470 (2001)
18. Andre, S., Ribeiro: Stochastic and delayed stochastic models of gene expression and regulation. *Mathematical Biosciences* 223(1), 1–11 (2010)
19. Savageau, M.A.: Comparison of classical and autogenous systems of regulation in inducible operons. *Nature* (1974)
20. Zhang, J., Sosonkina, M., Watson, L.T., Cao, Y.: Parallel solution of the chemical master equation. In: SpringSim (2009)

Assume-Guarantee Abstraction Refinement for Probabilistic Systems *

Anvesh Komuravelli¹, Corina S. Păsăreanu², and Edmund M. Clarke¹

¹ Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

² Carnegie Mellon Silicon Valley, NASA Ames, Moffett Field, CA, USA

Abstract. We describe an automated technique for assume-guarantee style checking of strong simulation between a system and a specification, both expressed as non-deterministic Labeled Probabilistic Transition Systems (LPTSes). We first characterize counterexamples to strong simulation as *stochastic trees* and show that simpler structures are insufficient. Then, we use these trees in an abstraction refinement algorithm that computes the assumptions for assume-guarantee reasoning as conservative LPTS abstractions of some of the system components. The abstractions are automatically refined based on tree counterexamples obtained from failed simulation checks with the remaining components. We have implemented the algorithms for counterexample generation and assume-guarantee abstraction refinement and report encouraging results.

1 Introduction

Probabilistic systems are increasingly used for the formal modeling and analysis of a wide variety of systems ranging from randomized communication and security protocols to nanoscale computers and biological processes. Probabilistic model checking is an automatic technique for the verification of such systems against formal specifications [2]. However, as in the classical non-probabilistic case [7], it suffers from the *state explosion* problem, where the state space of a concurrent system grows exponentially in the number of its components.

Assume-guarantee style compositional techniques [18] address this problem by decomposing the verification of a system into that of its smaller components and composing back the results, without verifying the whole system directly. When checking individual components, the method uses *assumptions* about the components' environments and then, discharges them on the rest of the system. For a system of two components, such reasoning is captured by the following simple assume-guarantee rule.

$$\frac{1 : L_1 \parallel A \preceq P \quad 2 : L_2 \preceq A}{L_1 \parallel L_2 \preceq P} (\text{ASYM})$$

* This research was sponsored by DARPA META II, GSRC, NSF, SRC, GM, ONR under contracts FA8650-10C-7079, 1041377 (Princeton University), CNS0926181/CNS0931985, 2005TJ1366, GMCMUCRLNV301, N000141010188, respectively, and the CMU-Portugal Program.

Here L_1 and L_2 are system components, P is a specification to be satisfied by the composite system and A is an assumption on L_1 's environment, to be discharged on L_2 . Several other such rules have been proposed, some of them involving symmetric [19] or circular [8, 19, 16] reasoning. Despite its simplicity, rule ASYM has been proven the most effective in practice and studied extensively [19, 4, 11], mostly in the context of non-probabilistic reasoning.

We consider here the *automated assume-guarantee* style compositional verification of *Labeled Probabilistic Transition Systems* (LPTSes), whose transitions have both probabilistic and non-deterministic behavior. The verification is performed using the rule ASYM where L_1 , L_2 , A and P are LPTSes and the conformance relation \preceq is instantiated with *strong simulation* [20]. We chose strong simulation for the following reasons. Strong simulation is a decidable, well studied relation between specifications and implementations, both for non-probabilistic [17] and probabilistic [20] systems. A method to help scale such a check is of a natural interest. Furthermore, rule ASYM is both sound and complete for this relation. Completeness is obtained trivially by replacing A with L_2 but is essential for full automation (see Section 5). One can argue that strong simulation is too fine a relation to yield suitably small assumptions. However, previous success in using strong simulation in non-probabilistic compositional verification [5] motivated us to consider it in a probabilistic setting as well. And we shall see that indeed we can obtain small assumptions for the examples we consider while achieving savings in time and memory (see Section 6).

The main challenge in automating assume-guarantee reasoning is to come up with such small assumptions satisfying the premises. In the non-probabilistic case, solutions to this problem have been proposed which use either automata learning techniques [19, 4] or abstraction refinement [12] and several improvements and optimizations followed. For probabilistic systems, techniques using automata learning have been proposed. They target *probabilistic reachability* checking and are not guaranteed to terminate due to incompleteness of the assume-guarantee rules [11] or to the undecidability of the conformance relation and learning algorithms used [10].

In this paper we propose a complete, fully automatic framework for the compositional verification of LPTSes with respect to simulation conformance. One fundamental ingredient of the framework is the use of *counterexamples* (from failed simulation checks) to iteratively refine inferred assumptions. Counterexamples are also extremely useful in general to help with debugging of discovered errors. However, to the best of our knowledge, the notion of a counterexample has not been previously formalized for strong simulation between probabilistic systems. As our first contribution we give a characterization of counterexamples to strong simulation as *stochastic trees* and an algorithm to compute them; we also show that simpler structures are insufficient in general (Section 3).

We then propose an assume-guarantee abstraction-refinement (AGAR) algorithm (Section 5) to automatically build the assumptions used in compositional reasoning. The algorithm follows previous work [12] which, however, was done in a non-probabilistic, trace-based setting. In our approach, A is maintained as a *conservative abstraction* of L_2 , *i.e.* an LPTS that simulates L_2 (hence, premise

2 holds by construction), and is iteratively refined based on tree counterexamples obtained from checking premise 1. The iterative process is guaranteed to terminate, with the number of iterations bounded by the number of states in L_2 . When L_2 itself is composed of multiple components, the second premise ($L_2 \preceq A$) is viewed as a new compositional check, generalizing the approach to $n \geq 2$ components. AGAR can be further applied to the case where the specification P is instantiated with a formula of a logic preserved by strong simulation, such as *safe-pCTL*.

We have implemented the algorithms for counterexample generation and for AGAR using JavaTM and Yices [9] and show experimentally that AGAR can achieve significantly better performance than non-compositional verification.

Other Related Work. Counterexamples to strong simulation have been characterized before as tree-shaped structures for the case of non-probabilistic systems [5] which we generalize to stochastic trees in Section 3 for the probabilistic case. Tree counterexamples have also been used in the context of a compositional framework that uses rule ASYM for checking strong simulation in the non-probabilistic case [4] and employs tree-automata learning to build deterministic assumptions.

AGAR is a variant of the well-known CounterExample Guided Abstraction Refinement (CEGAR) approach [6]. CEGAR has been adapted to probabilistic systems, in the context of probabilistic reachability [13] and safe-pCTL [3]. The CEGAR approach we describe in Section 4 is an adaptation of the latter. Both these works consider abstraction refinement in a monolithic, non-compositional setting. On the other hand, AGAR uses counterexamples from checking one component to refine the abstraction of another component.

2 Preliminaries

Labeled Probabilistic Transition Systems. Let S be a non-empty set. $Dist(S)$ is defined to be the set of discrete probability distributions over S . We assume that all the probabilities specified explicitly in a distribution are rationals in $[0, 1]$; there is no unique representation for all real numbers on a computer and floating-point numbers are essentially rationals. For $s \in S$, δ_s is the Dirac distribution on s , i.e. $\delta_s(s) = 1$ and $\delta_s(t) = 0$ for all $t \neq s$. For $\mu \in Dist(S)$, the *support* of μ , denoted $Supp(\mu)$, is defined to be the set $\{s \in S | \mu(s) > 0\}$ and for $T \subseteq S$, $\mu(T)$ stands for $\sum_{s \in T} \mu(s)$. The models we consider, defined below, have both probabilistic and non-deterministic behavior. Thus, there can be a non-deterministic choice between two probability distributions, even for the same action. Such modeling is mainly used for underspecification and moreover, the abstractions we consider (see Definition 8) naturally have this non-determinism. As we see below, the theory described does not become any simpler by disallowing non-deterministic choice for a given action (Lemmas 4 and 5).

Definition 1 (LPTS). A Labeled Probabilistic Transition System (LPTS) is a tuple $\langle S, s^0, \alpha, \tau \rangle$ where S is a set of states, $s^0 \in S$ is a distinguished start

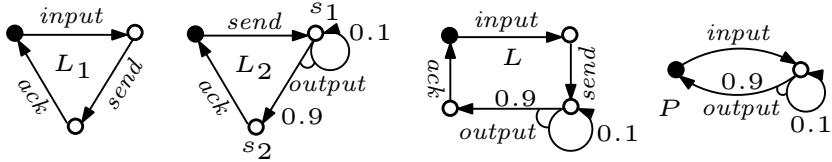


Fig. 1. Four reactive and fully-probabilistic LPTSes

state, α is a set of actions and $\tau \subseteq S \times \alpha \times \text{Dist}(S)$ is a probabilistic transition relation. For $s \in S$, $a \in \alpha$ and $\mu \in \text{Dist}(S)$, we denote $(s, a, \mu) \in \tau$ by $s \xrightarrow{a} \mu$ and say that s has a transition on a to μ .

An LPTS is called *reactive* if τ is a partial function from $S \times \alpha$ to $\text{Dist}(S)$ (i.e. at most one transition on a given action from a given state) and *fully-probabilistic* if τ is a partial function from S to $\alpha \times \text{Dist}(S)$ (i.e. at most one transition from a given state).

Figure 1 illustrates LPTSes. Throughout this paper, we use filled circles to denote start states in the pictorial representations of LPTSes. For the distribution $\mu = \{(s_1, 0.1), (s_2, 0.9)\}$, L_2 in the figure has the transition $s_1 \xrightarrow{\text{output}} \mu$. All the LPTSes in the figure are *reactive* as no state has more than one transition on a given action. They are also *fully-probabilistic* as no state has more than one transition. In the literature, an LPTS is also called a *simple probabilistic automaton* [20]. Similarly, a reactive (fully-probabilistic) LPTS is also called a (Labeled) *Markov Decision Process* (*Markov Chain*). Also, note that an LPTS with all the distributions restricted to Dirac distributions is the classical (non-probabilistic) *Labeled Transition System* (LTS); thus a *reactive* LTS corresponds to the standard notion of a *deterministic* LTS. For example, L_1 in Figure 1 is a reactive (or deterministic) LTS. We only consider finite state, finite alphabet and finitely branching (i.e. finitely many transitions from any state) LPTSes.

We are also interested in LPTSes with a tree structure, i.e. the start state is not in the support of any distribution and every other state is in the support of exactly one distribution. We call such LPTSes *stochastic trees* or simply, *trees*.

We use $\langle S_i, s_i^0, \alpha_i, \tau_i \rangle$ for an LPTS L_i and $\langle S_L, s_L^0, \alpha_L, \tau_L \rangle$ for an LPTS L . The following notation is used in Section 5.

Notation 1 For an LPTS L and an alphabet α with $\alpha_L \subseteq \alpha$, L^α stands for the LPTS $\langle S_L, s_L^0, \alpha, \tau_L \rangle$.

Let L_1 and L_2 be two LPTSes and $\mu_1 \in \text{Dist}(S_1)$, $\mu_2 \in \text{Dist}(S_2)$.

Definition 2 (Product [20]). The product of μ_1 and μ_2 , denoted $\mu_1 \otimes \mu_2$, is a distribution in $\text{Dist}(S_1 \times S_2)$, such that $\mu_1 \otimes \mu_2 : (s_1, s_2) \mapsto \mu_1(s_1) \cdot \mu_2(s_2)$.

Definition 3 (Composition [20]). The parallel composition of L_1 and L_2 , denoted $L_1 \parallel L_2$, is defined as the LPTS $\langle S_1 \times S_2, (s_1^0, s_2^0), \alpha_1 \cup \alpha_2, \tau \rangle$ where $((s_1, s_2), a, \mu) \in \tau$ iff

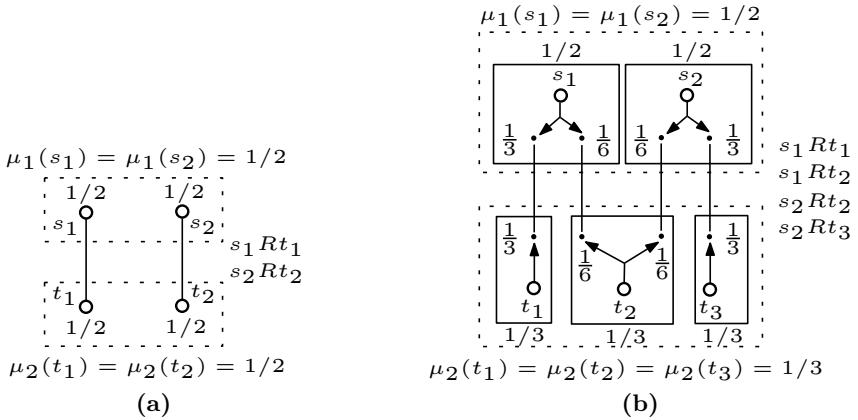


Fig. 2. Explaining $\mu_1 \sqsubseteq_R \mu_2$ by means of splitting (indicated by arrows) and matching (indicated by solid lines) the probabilities

1. $s_1 \xrightarrow{a} \mu_1, s_2 \xrightarrow{a} \mu_2$ and $\mu = \mu_1 \otimes \mu_2$, or
2. $s_1 \xrightarrow{a} \mu_1$, $a \notin \alpha_2$ and $\mu = \mu_1 \otimes \delta_{s_2}$, or
3. $a \notin \alpha_1, s_2 \xrightarrow{a} \mu_2$ and $\mu = \delta_{s_1} \otimes \mu_2$.

For example, in Figure 2, L is the composition of L_1 and L_2 .

Strong Simulation. For two LTSes, a pair of states belonging to a strong simulation relation depends on whether certain other pairs of successor states also belong to the relation [17]. For LPTSES, one has successor *distributions* instead of successor states; a pair of states belonging to a strong simulation relation R should now depend on whether certain other pairs in the *supports* of the successor distributions also belong to R . Therefore we define a binary relation on distributions, \sqsubseteq_R , which depends on the relation R between states. Intuitively, two distributions can be related if we can pair the states in their support sets, the pairs contained in R , *matching all* the probabilities under the distributions.

Consider an example with sRt and the transitions $s \xrightarrow{a} \mu_1$ and $t \xrightarrow{a} \mu_2$ with μ_1 and μ_2 as in Figure 2(a). In this case, one easy way to match the probabilities is to pair s_1 with t_1 and s_2 with t_2 . This is sufficient if s_1Rt_1 and s_2Rt_2 also hold, in which case, we say that $\mu_1 \sqsubseteq_R \mu_2$. However, such a direct matching may not be possible in general, as is the case in Figure 2(b). One can still obtain a matching by *splitting* the probabilities under the distributions in such a way that one can then directly match the probabilities as in Figure 2(a). Now, if s_1Rt_1 , s_1Rt_2 , s_2Rt_2 and s_2Rt_3 also hold, we say that $\mu_1 \sqsubseteq_R \mu_2$. Note that there can be more than one possible splitting. This is the central idea behind the following definition where the splitting is achieved by a *weight function*. Let $R \subseteq S_1 \times S_2$.

Definition 4 ([20]). $\mu_1 \sqsubseteq_R \mu_2$ iff there is a weight function $w : S_1 \times S_2 \rightarrow \mathbb{Q} \cap [0, 1]$ such that

1. $\mu_1(s_1) = \sum_{s_2 \in S_2} w(s_1, s_2)$ for all $s_1 \in S_1$,

2. $\mu_2(s_2) = \sum_{s_1 \in S_1} w(s_1, s_2)$ for all $s_2 \in S_2$,
3. $w(s_1, s_2) > 0$ implies $s_1 R s_2$ for all $s_1 \in S_1$, $s_2 \in S_2$.

$\mu_1 \sqsubseteq_R \mu_2$ can be checked by computing the maxflow in an appropriate network and checking if it equals 1.0 [1]. If $\mu_1 \sqsubseteq_R \mu_2$ holds, w in the above definition is one such maxflow function. As explained above, $\mu_1 \sqsubseteq_R \mu_2$ can be understood as *matching* all the probabilities (after splitting appropriately) under μ_1 and μ_2 . Considering $\text{Supp}(\mu_1)$ and $\text{Supp}(\mu_2)$ as two partite sets, this is the weighted analog of saturating a partite set in bipartite matching, giving us the following analog of the well-known Hall's Theorem for saturating $\text{Supp}(\mu_1)$.

Lemma 1 ([21]). $\mu_1 \sqsubseteq_R \mu_2$ iff for every $S \subseteq \text{Supp}(\mu_1)$, $\mu_1(S) \leq \mu_2(R(S))$.

It follows that when $\mu_1 \not\sqsubseteq_R \mu_2$, there exists a witness $S \subseteq \text{Supp}(\mu_1)$ such that $\mu_1(S) > \mu_2(R(S))$. For example, if $R(s_2) = \emptyset$ in Figure 2(a), its probability $\frac{1}{2}$ under μ_1 cannot be matched and $S = \{s_2\}$ is a witness subset.

Definition 5 (Strong Simulation [20]). R is a strong simulation iff for every $s_1 R s_2$ and $s_1 \xrightarrow{a} \mu_1^a$ there is a μ_2^a with $s_2 \xrightarrow{a} \mu_2^a$ and $\mu_1^a \sqsubseteq_R \mu_2^a$.

For $s_1 \in S_1$ and $s_2 \in S_2$, s_2 strongly simulates s_1 , denoted $s_1 \preceq s_2$, iff there is a strong simulation T such that $s_1 T s_2$. L_2 strongly simulates L_1 , also denoted $L_1 \preceq L_2$, iff $s_1^0 \preceq s_2^0$.

When checking a specification P of a system L with $\alpha_P \subset \alpha_L$, we implicitly assume that P is *completed* by adding Dirac self-loops on each of the actions in $\alpha_L \setminus \alpha_P$ from every state before checking $L \preceq P$. For example, $L \preceq P$ in Figure 1 assuming that P is completed with $\{\text{send}, \text{ack}\}$. Checking $L_1 \preceq L_2$ is decidable in polynomial time [1,21] and can be performed with a greatest fixed point algorithm that computes the coarsest simulation between L_1 and L_2 . The algorithm uses a relation variable R initialized to $S_1 \times S_2$ and checks the condition in Definition 5 for every pair in R , iteratively, removing any violating pairs from R . The algorithm terminates when a fixed point is reached showing $L_1 \preceq L_2$ or when the pair of initial states is removed showing $L_1 \not\preceq L_2$. If $n = \max(|S_1|, |S_2|)$ and $m = \max(|\tau_1|, |\tau_2|)$, the algorithm takes $O((mn^6 + m^2n^3)/\log n)$ time and $O(mn + n^2)$ space [1]. Several optimizations exist [21] but we do not consider them here, for simplicity.

We do consider a specialized algorithm for the case that L_1 is a tree which we use during abstraction refinement (Sections 4 and 5). It initializes R to $S_1 \times S_2$ and is based on a bottom-up traversal of L_1 . Let $s_1 \in S_1$ be a non-leaf state during such a traversal and let $s_1 \xrightarrow{a} \mu_1$. For every $s_2 \in S_2$, the algorithm checks if there exists $s_2 \xrightarrow{a} \mu_2$ with $\mu_1 \sqsubseteq_R \mu_2$ and removes (s_1, s_2) from R , otherwise, where R is the current relation. This constitutes an iteration in the algorithm. The algorithm terminates when (s_1^0, s_2^0) is removed from R or when the traversal ends. Correctness is not hard to show and we skip the proof.

Lemma 2 ([20]). \preceq is a preorder (i.e. reflexive and transitive) and is compositional, i.e. if $L_1 \preceq L_2$ and $\alpha_2 \subseteq \alpha_1$, then for every LPTS L , $L_1 \| L \preceq L_2 \| L$.

Finally, we show the *soundness* and *completeness* of the rule ASYM. The rule is *sound* if the conclusion holds whenever there is an A satisfying the premises. And the rule is *complete* if there is an A satisfying the premises whenever the conclusion holds.

Theorem 1. *For $\alpha_A \subseteq \alpha_2$, the rule ASYM is sound and complete.*

Proof. Soundness follows from Lemma 2. Completeness follows trivially by replacing A with L_2 . \square

3 Counterexamples to Strong Simulation

Let L_1 and L_2 be two LPTSes. We characterize a counterexample to $L_1 \preceq L_2$ as a tree and show that any simpler structure is not sufficient in general. We first describe counterexamples via a simple language-theoretic characterization.

Definition 6 (Language of an LPTS). *Given an LPTS L , we define its language, denoted $\mathcal{L}(L)$, as the set $\{L' | L' \text{ is an LPTS and } L' \preceq L\}$.*

Lemma 3. $L_1 \preceq L_2$ iff $\mathcal{L}(L_1) \subseteq \mathcal{L}(L_2)$.

Proof. Necessity follows trivially from the transitivity of \preceq and sufficiency follows from the reflexivity of \preceq which implies $L_1 \in \mathcal{L}(L_1)$. \square

Thus, a counterexample C can be defined as follows.

Definition 7 (Counterexample). *A counterexample to $L_1 \preceq L_2$ is an LPTS C such that $C \in \mathcal{L}(L_1) \setminus \mathcal{L}(L_2)$, i.e. $C \preceq L_1$ but $C \not\preceq L_2$.*

Now, L_1 itself is a trivial choice for C but it does not give any more useful information than what we had before checking the simulation. Moreover, it is preferable to have C with a special and simpler structure rather than a general LPTS as it helps in a more efficient counterexample analysis, wherever it is used. When the LPTSes are restricted to LTSes, a *tree-shaped* LTS is known to be sufficient as a counterexample [5]. Based on a similar intuition, we show that a stochastic tree is sufficient as a counterexample in the probabilistic case.

Theorem 2. *If $L_1 \not\preceq L_2$, there is a tree which serves as a counterexample.*

Proof. We only give a brief sketch of a constructive proof here. Counterexample generation is based on the coarsest strong simulation computation from Section 2. By induction on the size of the current relation R , we show that there is a tree counterexample to $s_1 \preceq s_2$ whenever (s_1, s_2) is removed from R . We only consider the inductive case here. The pair is removed because there is a transition $s_1 \xrightarrow{a} \mu_1$ but for every $s_2 \xrightarrow{a} \mu$, $\mu_1 \not\sqsubseteq_R \mu$ i.e. there exists $S_1^\mu \subseteq \text{Supp}(\mu_1)$ such that $\mu_1(S_1^\mu) > \mu(R(S_1^\mu))$. Such an S_1^μ can be found using Algorithm 1. Now, no pair in $S_1^\mu \times (\text{Supp}(\mu) \setminus R(S_1^\mu))$ is in R . By induction hypothesis, a counterexample tree exists for each such pair. A counterexample to $s_1 \preceq s_2$ is built using μ_1 and all these other trees. \square

Algorithm 1. Finding $T \subseteq S_1$ such that $\mu_1(T) > \mu(R(T))$.

Given $\mu_1 \in \text{Dist}(S_1)$, $\mu \in \text{Dist}(S_2)$, $R \subseteq S_1 \times S_2$ with $\mu_1 \not\sqsubseteq_R \mu$.

- 1: let f be a maxflow function for the flow network corresponding to μ_1 and μ
 - 2: find $s_1 \in S_1$ with $\mu_1(s_1) > \sum_{s_2 \in S_2} f(s_1, s_2)$ and let $T = \{s_1\}$
 - 3: **while** $\mu_1(T) \leq \mu(R(T))$ **do**
 - 4: $T \leftarrow \{s_1 \in S_1 \mid \exists s_2 \in R(T) : f(s_1, s_2) > 0\}$
 - 5: **end while**
 - 6: **return** T
-

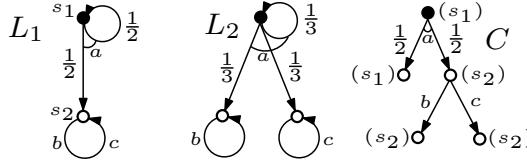


Fig. 3. C is a counterexample to $L_1 \preceq L_2$

For an illustration, see Figure 3 where C is a counterexample to $L_1 \preceq L_2$. Algorithm 1 is also analogous to the one used to find a subset failing Hall's condition in Graph Theory and can easily be proved correct. We obtain the following complexity bounds.

Theorem 3. *Deciding $L_1 \preceq L_2$ and obtaining a tree counterexample takes $O(mn^6 + m^2n^3)$ time and $O(mn + n^2)$ space where $n = \max(|S_{L_1}|, |S_{L_2}|)$ and $m = \max(|\tau_1|, |\tau_2|)$.*

Note that the obtained counterexample is essentially a finite *tree execution* of L_1 . That is, there is a total mapping $M : S_C \rightarrow S_1$ such that for every transition $c \xrightarrow{a} \mu_c$ of C , there exists $M(c) \xrightarrow{a} \mu_1$ such that M restricted to $\text{Supp}(\mu_c)$ is an injection and for every $c' \in \text{Supp}(\mu_c)$, $\mu_c(c') = \mu_1(M(c'))$. M is also a strong simulation. We call such a mapping an *execution mapping from C to L_1* . Figure 3 shows an execution mapping in brackets beside the states of C . We therefore have the following corollary.

Corollary 1. *If L_1 is reactive and $L_1 \not\preceq L_2$, there is a reactive tree which serves as a counterexample.*

The following two lemmas show that (reactive) trees are the simplest structured counterexamples.

Lemma 4. *There exist reactive LPTSes R_1 and R_2 such that $R_1 \not\preceq R_2$ and no counterexample is fully-probabilistic.*

Thus, if L_1 is reactive, a reactive tree is the simplest structure for a counterexample to $L_1 \preceq L_2$. This is surprising, since the non-probabilistic counterpart of a

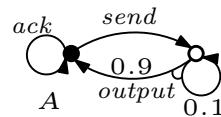
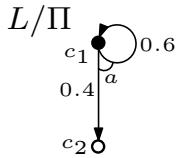
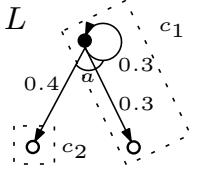


Fig. 4. An LPTS L , partition $\Pi = \{c_1, c_2\}$ and the quotient L/Π

Fig. 5. An assumption for L_1 , L_2 and P in Figure 11

fully-probabilistic LPTS is a trace of actions and it is known that trace inclusion coincides with simulation conformance between reactive (*i.e.* deterministic) LPTSes. If there is no such restriction on L_1 , one may ask if a reactive LPTS suffices as a counterexample to $L_1 \preceq L_2$. That is not the case either, as the following lemma shows.

Lemma 5. *There exist an LPTS L and a reactive LPTS R such that $L \not\preceq R$ and no counterexample is reactive.*

4 CEGAR for Checking Strong Simulation

Now that the notion of a counterexample has been formalized, we describe a CounterExample Guided Abstraction Refinement (CEGAR) approach [6] to check $L \preceq P$ where L and P are LPTSes and P stands for a *specification* of L . We will use this approach to describe AGAR in the next section.

Abstractions for L are obtained using a quotient construction from a *partition* Π of S_L . We let Π also denote the corresponding set of equivalence classes and given an arbitrary $s \in S$, let $[s]_\Pi$ denote the equivalence class containing s . The quotient is an adaptation of the usual construction in the non-probabilistic case.

Definition 8 (Quotient LPTS). *Given a partition Π of S_L , define the quotient LPTS, denoted L/Π , as the LPTS $\langle \Pi, [s^0_L]_\Pi, \alpha_L, \tau \rangle$ where $(c, a, \mu_i) \in \tau$ iff $(s, a, \mu) \in \tau_L$ for some $s \in S_L$ with $s \in c$ and $\mu_i(c') = \sum_{t \in c'} \mu(t)$ for all $c' \in \Pi$.*

As the abstractions are built from an explicit representation of L , this is not immediately useful. But, as we will see in Sections 5 and 6, this becomes very useful when adapted to the assume-guarantee setting.

Figure 4 shows an example quotient. Note that $L \preceq L/\Pi$ for any partition Π of S_L , with the relation $R = \{(s, c) | s \in c, c \in \Pi\}$ as a strong simulation.

CEGAR for LPTSes is sketched in Algorithm 2. It maintains an abstraction A of L , initialized to the quotient for the coarsest partition, and iteratively refines A based on the counterexamples obtained from the simulation check against P until a partition whose corresponding quotient conforms to P w.r.t. \preceq is obtained, or a real counterexample is found. In the following, we describe how to analyze if a counterexample is *spurious*, due to abstraction, and how to refine the abstraction in case it is (lines 4 – 6). Our analysis is an adaptation of an existing one for counterexamples which are arbitrary *sub-structures* of A [3];

Algorithm 2. CEGAR for LPTSes: checks $L \preceq P$

```

1:  $A \leftarrow L/\Pi$ , where  $\Pi$  is the coarsest partition of  $S_L$ 
2: while  $A \not\preceq P$  do
3:   obtain a counterexample  $C$ 
4:    $(spurious, A') \leftarrow analyzeAndRefine(C, A, L)$  {see text}
5:   if spurious then
6:      $A \leftarrow A'$ 
7:   else
8:     return counterexample  $C$ 
9:   end if
10: end while
11: return  $L \preceq P$  holds

```

while our tree counterexamples have an execution mapping to A , they are not necessarily sub-structures of A .

Analysis and Refinement (*analyzeAndRefine()*). Assume that Π is a partition of S_L such that $A = L/\Pi$ and $A \not\preceq P$. Let C be a tree counterexample obtained by the algorithm described in Section 3, i.e. $C \preceq A$ but $C \not\preceq P$. As described in Section 3, there is an *execution mapping* $M : S_C \rightarrow S_A$ which is also a strong simulation. Let $R_M \subseteq S_C \times S_L$ be $\{(s_1, s_2) | s_1 M [s_2]_\Pi\}$. Our refinement strategy tries to obtain the coarsest strong simulation between C and L contained in R_M , using the specialized algorithm for trees described in Section 2 with R_M as the initial candidate. Let R and R_{old} be the candidate relations at the end of the current and the previous iterations, respectively, and let $s_1 \xrightarrow{a} \mu_1$ be the transition in C considered by the algorithm in the current iteration. (R_{old} is undefined initially.) The strategy refines a state when one of the following two cases happens before termination and otherwise, returns C as a *real* counterexample.

1. $R(s_1) = \emptyset$. There are two possible reasons for this case. One is that the states in $Supp(\mu_1)$ are not related, by R , to enough number of states in S_L (i.e. μ_1 is *spurious*) and (the images under M of) all the states in $Supp(\mu_1)$ are candidates for refinement. The other possible reason is the branching (more than one transition) from s_1 where no state in $R_M(s_1)$ can *simulate all* the transitions of s_1 and $M(s_1)$ is a candidate for refinement.
2. $M(s_1) = [s_L^0]_\Pi$, $s_L^0 \in R_{old}(s_1) \setminus R(s_1)$ and $R(s_1) \neq \emptyset$, i.e. $M(s_1)$ is the initial state of A but s_1 is no longer related to s_L^0 by R . Here, $M(s_1)$ is a candidate for refinement.

In case 1, our refinement strategy first tries to split the equivalence class $M(s_1)$ into $R_{old}(s_1)$ and the rest and then, for every state $s \in Supp(\mu_1)$, tries to split the equivalence class $M(s)$ into $R_{old}(s)$ and the rest, unless $M(s) = M(s_1)$ and $M(s_1)$ has already been split. And in case 2, the strategy splits the equivalence class $M(s_1)$ into $R_{old}(s_1) \setminus R(s_1)$ and the rest. It follows from the two cases that if C is declared real, then $C \preceq L$ with the final R as a strong simulation between

C and L and hence, C is a counterexample to $L \preceq P$. The following lemma shows that the refinement strategy always leads to progress.

Lemma 6. *The above refinement strategy always results in a strictly finer partition $\Pi' < \Pi$.*

5 Assume-Guarantee Abstraction Refinement

We now describe our approach to Assume-Guarantee Abstraction Refinement (AGAR) for LPTSes. The approach is similar to CEGAR from the previous section with the notable exception that counterexample analysis is performed in an assume guarantee style: a counterexample obtained from checking one component is used to refine the abstraction of a different component.

Given LPTSes L_1 , L_2 and P , the goal is to check $L_1 \parallel L_2 \preceq P$ in an assume-guarantee style, using rule ASYM. The basic idea is to maintain A in the rule as an abstraction of L_2 , *i.e.* the second premise holds for free throughout, and to check only the first premise for every A generated by the algorithm. As in CEGAR, we restrict A to the quotient for a partition of S_2 . If the first premise holds for an A , then $L_1 \parallel L_2 \preceq P$ also holds, by the soundness of the rule. Otherwise, the obtained counterexample C is analyzed to see whether it indicates a real error or it is spurious, in which case A is refined (as described in detail below). Algorithm 3 sketches the AGAR loop.

For an example, A in Figure 5 shows the final assumption generated by AGAR for the LPTSes in Figure 11 (after one refinement).

Algorithm 3. AGAR for LPTSes: checks $L_1 \parallel L_2 \preceq P$

```

1:  $A \leftarrow$  coarsest abstraction of  $L_2$ 
2: while  $L_1 \parallel A \not\preceq P$  do
3:   obtain a counterexample  $C$ 
4:   obtain projections  $C \upharpoonright_{L_1}$  and  $C \upharpoonright_A$ 
5:    $(spurious, A') \leftarrow analyzeAndRefine(C \upharpoonright_A, A, L_2)$ 
6:   if  $spurious$  then
7:      $A \leftarrow A'$ 
8:   else
9:     return counterexample  $C$ 
10:  end if
11: end while
12: return  $L_1 \parallel L_2 \preceq P$  holds

```

Analysis and Refinement. The counterexample analysis is performed compositionally, using the *projections* of C onto L_1 and A . As there is an *execution mapping* from C to $L_1 \parallel A$, these projections are the *contributions* of L_1 and A towards C in the composition. We denote these projections by $C \upharpoonright_{L_1}$ and $C \upharpoonright_A$,

respectively. In the non-probabilistic case, these are obtained by simply projecting C onto the respective alphabets. In the probabilistic scenario, however, composition changes the probabilities in the distributions (Definition 2) and alphabet projection is insufficient. For this reason, we additionally record the individual distributions of the LPTSes responsible for a product distribution while performing the composition. Thus, projections $C \upharpoonright_{L_1}$ and $C \upharpoonright_A$ can be obtained using this auxiliary information. Note that there is a natural *execution mapping* from $C \upharpoonright_A$ to A and from $C \upharpoonright_{L_1}$ to L_1 . We can then employ the analysis described in Section 4 between $C \upharpoonright_A$ and A , i.e. invoke *analyzeAndRefine*($C \upharpoonright_A, A, L_2$) to determine if $C \upharpoonright_A$ (and hence, C) is spurious and to refine A in case it is. Otherwise, $C \upharpoonright_A \preceq L_2$ and hence, $(C \upharpoonright_A)^{\alpha_2} \preceq L_2$. Together with $(C \upharpoonright_{L_1})^{\alpha_1} \preceq L_1$ this implies $(C \upharpoonright_{L_1})^{\alpha_1} \parallel (C \upharpoonright_A)^{\alpha_2} \preceq L_1 \parallel L_2$ (Lemma 2). As $C \preceq (C \upharpoonright_{L_1})^{\alpha_1} \parallel (C \upharpoonright_A)^{\alpha_2}$, C is then a *real counterexample*. Thus, we have the following result.

Theorem 4 (Correctness and Termination). *Algorithm AGAR always terminates with at most $|S_2| - 1$ refinements and $L_1 \parallel L_2 \not\preceq P$ if and only if the algorithm returns a real counterexample.*

Proof. Correctness: AGAR terminates when either Premise 1 is satisfied by the current assumption (line 12) or when a counterexample is returned (line 9). In the first case, we know that Premise 2 holds by construction and since ASYM is sound (Theorem 1) it follows that indeed $L_1 \parallel L_2 \preceq P$. In the second case, the counterexample returned by AGAR is real (see above) showing that $L_1 \parallel L_2 \not\preceq P$.

Termination: AGAR iteratively refines the abstraction until the property holds or a real counterexample is reported. Abstraction refinement results in a finer partition (Lemma 6) and thus it is guaranteed to terminate since in the worst case A converges to L_2 which is finite state. Since rule ASYM is trivially complete for L_2 (proof of Theorem 1) it follows that AGAR will also terminate, and the number of refinements is bounded by $|S_2| - 1$. \square

In practice, we expect AGAR to terminate earlier than in $|S_2| - 1$ steps, with an assumption smaller than L_2 . AGAR will terminate as soon as it finds an assumption that satisfies the premises or that helps exhibit a real counterexample. Note also that, although AGAR uses an explicit representation for the individual components, it never builds $L_1 \parallel L_2$ directly (except in the worst-case) keeping the cost of verification low.

Reasoning with $n \geq 2$ Components. So far, we have discussed compositional verification in the context of two components L_1 and L_2 . This reasoning can be generalized to $n \geq 2$ components using the following (sound and complete) rule.

$$\frac{1 : L_1 \parallel A_1 \preceq P \quad 2 : L_2 \parallel A_2 \preceq A_1 \quad \dots \quad n : L_n \preceq A_{n-1}}{\parallel_{i=1}^n L_i \preceq P} (\text{ASYM-N})$$

The rule enables us to overcome the *intermediate state explosion* that may be associated with two-way decompositions (when the subsystems are larger than the entire system). The AGAR algorithm for this rule involves the creation of

$n - 1$ nested instances of AGAR for two components, with the i th instance computing the assumption A_i for $(L_1 \parallel \dots \parallel L_i) \parallel (L_{i+1} \parallel A_{i+1}) \preceq P$. When the AGAR instance for A_{i-1} returns a counterexample C , for $1 < i \leq n - 1$, we need to analyze C for spuriousness and refine A_i in case it is. From Algorithm 3, C is returned only if $\text{analyzeAndRefine}(C \upharpoonright_{A_{i-1}}, A_{i-1}, L_i \parallel A_i)$ concludes that $C \upharpoonright_{A_{i-1}}$ is real (note that A_{i-1} is an abstraction of $L_i \parallel A_i$). From analyzeAndRefine in Section 4, this implies that the final relation R computed between the states of $C \upharpoonright_{A_{i-1}}$ and $L_i \parallel A_i$ is a strong simulation between them. It follows that, although $C \upharpoonright_{A_{i-1}}$ does not have an *execution mapping* to $L_i \parallel A_i$, we can naturally obtain a tree T using $C \upharpoonright_{A_{i-1}}$, via R , with such a mapping. Thus, we modify the algorithm to return $T \upharpoonright_{A_i}$ at line 9, instead of C , which can then be used to check for spuriousness and refine A_i . Note that when A_i is refined, all the A_j 's for $j < i$ need to be recomputed.

Compositional Verification of Logical Properties. AGAR can be further applied to automate assume-guarantee checking of properties ϕ written as formulae in a logic that is preserved by strong simulation such as the *weak-safety* fragment of probabilistic CTL (pCTL) [3] which also yield trees as counterexamples. The rule ASYM is both sound and complete for this logic (\models denotes property satisfaction) for $\alpha_A \subseteq \alpha_2$ with a proof similar to that of Theorem 11.

$$\frac{1 : L_1 \parallel A \models \phi \quad 2 : L_2 \preceq A}{L_1 \parallel L_2 \models \phi}$$

A can be computed as a conservative abstraction of L_2 and iteratively refined based on the tree counterexamples to premise 1, using the same procedures as before. The rule can be generalized to reasoning about $n \geq 2$ components as described above and also to richer logics with more general counterexamples adapting existing CEGAR approaches [3] to AGAR. We plan to further investigate this direction in the future.

6 Implementation and Results

Implementation. We implemented the algorithms for checking simulation (Section 2), for generating counterexamples (as in the proof of Lemma 2) and for AGAR (Algorithm 3) with ASYM and ASYM-N in JavaTM. We used the front-end of PRISM's [15] explicit-state engine to parse the models of the components described in PRISM's input language and construct LPTSes which were then handled by our implementation.

While the JavaTM implementation for checking simulation uses the greatest fixed point computation to obtain the coarsest strong simulation, we noticed that the problem of checking the existence of a strong simulation is essentially a constraint satisfaction problem. To leverage the efficient constraint solvers that exist today, we reduced the problem of checking simulation to an SMT problem with rational linear arithmetic as follows. For every pair of states, the constraint that the pair is in some strong simulation is simply the encoding of the condition

in Definition 5. For a relevant pair of distributions μ_1 and μ_2 , the constraint for $\mu_1 \sqsubseteq_R \mu_2$ is encoded by means of a weight function (as given by Definition 4) and the constraint for $\mu_1 \not\sqsubseteq_R \mu_2$ is encoded by means of a witness subset of $Supp(\mu_1)$ (as in Lemma 1), where R is the variable for the strong simulation. We use Yices (v1.0.29) [9] to solve the resulting SMT problem; a *real* variable in Yices input language is essentially a rational variable. There is no direct way to obtain a tree counterexample when the SMT problem is unsatisfiable. Therefore when the conformance fails, we obtain the *unsat core* from Yices, construct the *sub-structure* of L_1 (when we check $L_1 \preceq L_2$) from the constraints in the unsat core and check the conformance of this sub-structure against L_2 using the JavaTM implementation. This sub-structure is usually much smaller than L_1 and contains only the information necessary to expose the counterexample.

Results. We evaluated our algorithms using this implementation on several examples analyzed in previous work [11]. Some of these examples were created by introducing probabilistic failures into non-probabilistic models used earlier [19] while others were adapted from PRISM benchmarks [15]. The properties used previously were about *probabilistic reachability* and we had to create our own specification LPTSes after developing an understanding of the models. The models in all the examples satisfy the respective specifications. We briefly describe the models and the specifications below, all of which are available at <http://www.cs.cmu.edu/~akomurav/publications/agar/AGAR.html>.

CS₁ and *CS_N* model a *Client-Server* protocol with mutual exclusion having probabilistic failures in one or all of the N clients, respectively. The specifications describe the probabilistic failure behavior of the clients while hiding some of the actions as is typical in a high level design specification.

MER models a *resource arbiter* module of NASA's software for *Mars Exploration Rovers* which grants and rescinds shared resources for several users. We considered the case of two resources with varying number of users and probabilistic failures introduced in all the components. As in the above example, the specifications describe the probabilistic failure behavior of the users while hiding some of the actions.

SN models a wireless *Sensor Network* of one or more sensors sending data and messages to a process via a channel with a bounded buffer having probabilistic behavior in the components. Creating specification LPTSes for this example turned out to be more difficult than the above examples, and we obtained them by observing the system's runs and by manual abstraction.

Table 1 shows the results we obtained when ASYM and ASYM-N were compared with monolithic (non-compositional) conformance checking. $|X|$ stands for the number of states of an LPTS X . L stands for the whole system, P for the specification, L_M for the LPTS with the largest number of states built by composing LPTSes during the course of AGAR, A_M for the assumption with the largest number of states during the execution and L_c for the component with the largest number of states in ASYM-N. *Time* is in seconds and *Memory* is in megabytes. We also compared $|L_M|$ with $|L|$, as $|L_M|$ denotes the largest LPTS ever built

Table 1. AGAR vs monolithic verification.¹ Mem-out during model construction.

Example (param)	L	P	ASYM						ASYM-N						MONO	
			L ₁	L ₂	Time	Mem	L _M	A _M	L _c	Time	Mem	L _M	A _M	Time	Mem	
CS ₁ (5)	94	16	36	405	7.2	15.6	182	33	36	74.0	15.1	182	34	0.2	8.8	
CS ₁ (6)	136	19	49	1215	11.6	22.7	324	41	49	810.7	21.4	324	40	0.5	12.2	
CS ₁ (7)	186	22	64	3645	37.7	49.4	538	56	64	out	—	—	—	0.8	17.9	
CS _N (2)	34	15	25	9	0.7	7.1	51	7	9	2.4	6.8	40	25	0.1	5.9	
CS _N (3)	184	54	125	16	43.0	63.0	324	12	16	1.6k	109.6	372	125	14.8	37.9	
CS _N (4)	960	189	625	25	out	—	—	25	—	out	—	—	—	1.8k	667.5	
MER (3)	16k	12	278	1728	2.6	19.7	706	7	278	3.6	14.6	706	7	193.8	458.5	
MER (4)	120k	15	465	21k	15.0	53.9	2k	11	465	34.7	37.8	2k	11	out	—	
MER (5)	841k	18	700	250k	—	out ¹	—	700	257.8	65.5	3.3k	16	—	out ¹	—	
SN (1)	462	18	43	32	0.2	6.2	43	3	126	1.7	8.5	165	6	1.5	27.7	
SN (2)	7860	54	796	32	79.5	112.9	796	3	252	694.4	171.7	1.4k	21	4.7k	1.3k	
SN (3)	78k	162	7545	32	out	—	—	378	7.2k	528.8	1.4k	21	—	out	—	

by AGAR. Best figures, among ASYM, ASYM-N and MONO, for *Time*, *Memory* and LPTS sizes, are boldfaced. All the results were taken on a Fedora-10 64-bit machine running on an Intel® Core™2 Quad CPU of 2.83GHz and 4GB RAM. We imposed a 2GB upper bound on Java heap memory and a 2 hour upper bound on the running time. We observed that most of the time during AGAR was spent in checking the premises and an insignificant amount was spent for the composition and the refinement steps. Also, most of the memory was consumed by Yices. We tried several orderings of the components (the L_i 's in the rules) and report only the ones giving the best results.

While monolithic checking outperformed AGAR for *Client-Server*, there are significant time and memory savings for *MER* and *Sensor Network* where in some cases the monolithic approach ran out of resources (time or memory). One possible reason for AGAR performing worse for *Client-Server* is that $|L|$ is much smaller than $|L_1|$ or $|L_2|$. When compared to using ASYM, ASYM-N brings further memory savings in the case of *MER* and also time savings for *Sensor Network* with parameter 3 which could not finish in 2 hours when used with ASYM. As already mentioned, these models were analyzed previously with an assume-guarantee framework using learning from traces [1]. Although that approach uses a similar assume-guarantee rule (but instantiated to check *probabilistic reachability*) and the results have some similarity (e.g. *Client-Server* is similarly not handled well by the compositional approach), we can not directly compare it with AGAR as it considers a different class of properties.

7 Conclusion and Future Work

We described a complete, fully automated abstraction-refinement approach for assume-guarantee checking of strong simulation between LPTSes. The approach uses refinement based on counterexamples formalized as stochastic trees and it further applies to checking *safe-pCTL* properties. We showed experimentally the merits of the proposed technique. We plan to extend our approach to cases where the assumption A has a smaller alphabet than that of the component

it represents as this can potentially lead to further savings. Strong simulation would no longer work and one would need to use *weak* simulation [20], for which checking algorithms are unknown yet. We would also like to explore symbolic implementations of our algorithms, for increased scalability. As an alternative approach, we plan to build upon our recent work [14] on learning LPTSes to develop practical compositional algorithms and compare with AGAR.

Acknowledgments. We thank Christel Baier, Rohit Chadha, Lu Feng, Holger Hermanns, Marta Kwiatkowska, Joel Ouaknine, David Parker, Frits Vaandrager, Mahesh Viswanathan, James Worrell and Lijun Zhang for generously answering our questions related to this research. We also thank the anonymous reviewers for their suggestions and David Henriques for carefully reading an earlier draft.

References

1. Baier, C.: On Algorithmic Verification Methods for Probabilistic Systems. Habilitation thesis, Fakultät für Mathematik und Informatik, Univ. Mannheim (1998)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Chadha, R., Viswanathan, M.: A Counterexample-Guided Abstraction-Refinement Framework for Markov Decision Processes. TOCL 12(1), 1–49 (2010)
4. Chaki, S., Clarke, E., Sinha, N., Thati, P.: Automated Assume-Guarantee Reasoning for Simulation Conformance. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 534–547. Springer, Heidelberg (2005)
5. Chaki, S.J.: A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs. PhD thesis, Carnegie Mellon University (2005)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
8. de Alfaro, L., Henzinger, T.A., Jhala, R.: Compositional Methods for Probabilistic Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 351–365. Springer, Heidelberg (2001)
9. Dutertre, B., Moura, L.D.: The Yices SMT Solver. Technical report, SRI International (2006)
10. Feng, L., Han, T., Kwiatkowska, M., Parker, D.: Learning-Based Compositional Verification for Synchronous Probabilistic Systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 511–521. Springer, Heidelberg (2011)
11. Feng, L., Kwiatkowska, M., Parker, D.: Automated Learning of Probabilistic Assumptions for Compositional Reasoning. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 2–17. Springer, Heidelberg (2011)
12. Gheorghiu Bozaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated Assume-Guarantee Reasoning by Abstraction Refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
13. 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)

14. Komuravelli, A., Păsăreanu, C.S., Clarke, E.M.: Learning Probabilistic Systems from Tree Samples. In: LICS (to appear, 2012)
15. 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)
16. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Assume-Guarantee Verification for Probabilistic Systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 23–37. Springer, Heidelberg (2010)
17. Milner, R.: An Algebraic Definition of Simulation between Programs. Technical report, Stanford University (1971)
18. Pnueli, A.: In Transition from Global to Modular Temporal Reasoning about Programs. In: LMCS. NATO ASI, vol. 13, pp. 123–144. Springer (1985)
19. Păsăreanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to Divide and Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning. FMSD 32(3), 175–205 (2008)
20. Segala, R., Lynch, N.: Probabilistic Simulations for Probabilistic Processes. Nordic J. of Computing 2(2), 250–273 (1995)
21. Zhang, L.: Decision Algorithms for Probabilistic Simulations. PhD thesis, Universität des Saarlandes (2008)

Cross-Entropy Optimisation of Importance Sampling Parameters for Statistical Model Checking

Cyrille Jegourel, Axel Legay, and Sean Sedwards

INRIA Rennes - Bretagne Atlantique
`cyrille.jegourel,axel.legay,sean.sedwards}@inria.fr`

Abstract. Statistical model checking avoids the exponential growth of states associated with probabilistic model checking by estimating probabilities from multiple executions of a system and by giving results within confidence bounds. Rare properties are often important but pose a particular challenge for simulation-based approaches, hence a key objective for statistical model checking (SMC) is to reduce the number and length of simulations necessary to produce a result with a given level of confidence. *Importance sampling* can achieve this, however to maintain the advantages of SMC it is necessary to find good importance sampling distributions without considering the entire state space.

Here we present a simple algorithm that uses the notion of cross-entropy to find an optimal importance sampling distribution. In contrast to previous work, our algorithm uses a naturally defined low dimensional vector of parameters to specify this distribution and thus avoids the intractable explicit representation of a transition matrix. We show that our parametrisation leads to a unique optimum and can produce many orders of magnitude improvement in simulation efficiency. We demonstrate the efficacy of our methodology by applying it to models from reliability engineering and biochemistry.

1 Introduction

The need to provide accurate predictions about the behaviour of complex systems is increasingly urgent. With computational power becoming ever-more affordable and compact, computational systems are inevitably becoming increasingly concurrent, distributed and adaptive, creating a correspondingly increased burden to check that they function correctly. At the same time, users expect high performance and reliability, prompting the need for equally high performance analysis tools and techniques.

The most common method to ensure the correctness of a system is by testing it with a number of test cases having predicted outcomes that can highlight specific problems. Testing techniques have been effective discovering bugs in many industrial applications and have been incorporated into sophisticated tools [9]. Despite this, testing is limited by the need to hypothesise scenarios that may cause failure and the fact that a reasonable set of test cases is unlikely to cover

all possible eventualities; errors and modes of failure may remain undetected and quantifying the likelihood of failure using a series of test cases is difficult.

Model checking is a formal technique that verifies whether a system satisfies a property specified in temporal logic under all possible scenarios. In recognition of non-deterministic systems and the fact that a Boolean answer is not always useful, *probabilistic* model checking quantifies the probability that a system satisfies a property. In particular, ‘numerical’ (alternatively ‘exact’) probabilistic model checking offers precise and accurate analysis by exhaustively exploring the state space of non-deterministic systems and has been successfully applied to a wide variety of protocols, algorithms and systems. The result of this technique is the exact (within limits of numerical precision) probability that a system will satisfy a property of interest, however the exponential growth of the state space limits its applicability. The typical 10^8 state limit of exhaustive approaches usually represents an insignificant fraction of the state space of real systems that may have tens of orders of magnitude more states than the number of protons in the universe ($\sim 10^{80}$).

Under certain circumstances it is possible to guarantee the performance of a system by specifying it in such a way that (particular) faults are impossible. Compositional reasoning and various symmetry reduction techniques can also be used to combat state-space explosion, but in general the size, unpredictability and heterogeneity of real systems [2] make these techniques infeasible. Static analysis has also been highly successful in analysing and debugging software and other systems, although it cannot match the precision of quantitative analysis of dynamic properties achieved using probabilistic and stochastic temporal logic.

While the state space explosion problem is unlikely to ever be adequately solved for all systems, simulation-based approaches are becoming increasingly tractable due to the availability of high performance hardware and algorithms. In particular, statistical model checking (SMC) combines the simplicity of testing with the formality and precision of numerical model checking; the core idea being to create multiple independent execution traces of the system and individually verify whether they satisfy some given property. By modelling the executions as a Bernoulli random variable and using advanced statistical techniques, such as Bayesian inference [4] and hypothesis testing [27], the results are combined in an efficient manner to decide whether the system satisfies the property with some level of confidence, or to estimate the probability that it does. Knowing a result with less than 100% confidence is often sufficient in real applications, since the confidence bounds may be made arbitrarily tight. Moreover, SMC may offer the only feasible means of quantifying the performance of many complex systems. Evidence of this is that SMC has been used to find bugs in large, heterogeneous aircraft systems [2]. Notable SMC platforms include APMC [11], YMER [28] and VESTA [23]. Moreover, well-established numerical model checkers, such as PRISM [17] and UPPAAL [3], are now also including SMC engines.

A key challenge facing SMC is to reduce the length (steps and cpu time) and number of simulation traces necessary to achieve a result with given confidence. The current proliferation of parallel computer architectures (multiple cpu cores,

grids, clusters, clouds and general purpose computing on graphics processors, etc.) favours SMC by making the production of multiple independent simulation runs relatively easy. Despite this, certain models still require a large number of simulation steps to verify a property and it is thus necessary to make simulation as efficient as possible. Rare properties pose a particular problem for simulation-based approaches, since they are not only difficult to observe (by definition) but their probability is difficult to bound [10].

The term ‘rare event’ is ubiquitous in the literature, but here we specifically consider rare *properties* defined in temporal logic. This distinguishes rare states from rare paths that may or may not contain rare states. In what follows we consider discrete space Markov models and present a simple algorithm to find an optimal set of importance sampling parameters, using the concept of minimum cross-entropy [16][25]. Our parametrisation arises naturally from the syntactic description of the model and thus constitutes a low dimensional vector in comparison to the state space of the model. We show that this parametrisation has a unique optimum and demonstrate its effectiveness on reliability and (bio)chemical models. We describe the advantages and potential pitfalls of our approach and highlight areas for future research.

2 Importance Sampling

Our goal is to estimate the probability of a property by simulation and bound the error of our estimation. When the property is not rare there are standard bounding formulae (e.g., the Chernoff and Hoeffding bounds [4][12]) that relate absolute error, confidence and the required number of simulations to achieve them, *independent* of the probability of the property. As the property becomes rarer, however, absolute error ceases to be useful and it is necessary to consider relative error, defined as the standard deviation of the estimate divided by its expectation. With Monte Carlo simulation relative error is unbounded with increasing rarity [21], but it is possible to bound the error by means of importance sampling [24][10].

Importance sampling is a technique that can improve the efficiency of simulating rare events and has been receiving considerable interest of late in the field of SMC (e.g., [5][1]). It works by simulating under an (importance sampling) distribution that makes a property more likely to be seen and then uses the results to calculate the probability under the original distribution by compensating for the differences. The concept arose from work on the ‘Monte Carlo method’ [18] in the Manhattan project during the 1940s and was originally used to quantify the performance of materials and solve otherwise intractable analytical problems with limited computer power (see, e.g., [15]). For importance sampling to be effective it is necessary to define a ‘good’ importance sampling distribution: (i) the property of interest must be seen frequently in simulations and (ii) the distribution of the paths that satisfy the property in the importance sampling distribution must be as close as possible to the distribution of the same paths in the original distribution (up to a normalising factor). The literature in this

field sometimes uses the term ‘zero variance’ to describe an optimal importance sampling distribution, referring to the fact that with an optimum importance sampling distribution all simulated paths satisfy the property and the estimator has zero variance. It is important to note, however, that a sub-optimal distribution may meet requirement (i) without necessarily meeting requirement (ii). Failure to consider (ii) can result in gross errors and overestimates of confidence (e.g. a distribution that simulates just one path that satisfies the given property). The algorithm we present in Section 3 addresses both (i) and (ii).

Importance sampling schemes fall into two broad categories: *state dependent tilting* and *state independent tilting* [5]. State dependent tilting refers to importance sampling distributions that individually bias (‘tilt’) every transition probability in the system. State independent tilting refers to importance sampling distributions that change classes of transition probabilities, independent of state. The former offers greatest precision but is infeasible for large models. The latter is more tractable but may not produce good importance sampling distributions. Our approach is a kind of *parametrised tilting* that potentially affects all transitions differently, but does so according to a set of parameters.

2.1 Estimators

Let Ω be a probability space of paths, with f a probability density function over Ω and $z(\omega) \in \{0, 1\}$ a function indicating whether a path ω satisfies some property ϕ . In the present context, z is defined by a formula of an arbitrary temporal logic over execution traces. The probability γ that ϕ occurs in a path is then given by

$$\gamma = \int_{\Omega} z(\omega) f(\omega) d\omega \quad (1)$$

and the standard Monte Carlo estimator of γ is given by

$$\tilde{\gamma} = \frac{1}{N_{MC}} \sum_{i=1}^{N_{MC}} z(\omega_i)$$

N_{MC} denotes the number of simulations used by the Monte Carlo estimator and ω_i is sampled according to f . Note that $z(\omega_i)$ is effectively the realisation of a Bernoulli random variable with parameter γ . Hence $\text{Var}(\tilde{\gamma}) = \gamma(1 - \gamma)$ and for $\gamma \rightarrow 0$, $\text{Var}(\tilde{\gamma}) \approx \gamma$. Let f' be another probability density function over Ω , absolutely continuous with zf , then Equation (1) can be written

$$\gamma = \int_{\Omega} z(\omega) \frac{f(\omega)}{f'(\omega)} f'(\omega) d\omega$$

$L = f/f'$ is the *likelihood ratio* function, so

$$\gamma = \int_{\Omega} L(\omega) z(\omega) f'(\omega) d\omega \quad (2)$$

We can thus estimate γ by simulating under f' and compensating by L :

$$\hat{\gamma} = \frac{1}{N_{\text{IS}}} \sum_{i=1}^{N_{\text{IS}}} L(\omega_i) z(\omega_i)$$

N_{IS} denotes the number of simulations used by the importance sampling estimator. The goal of importance sampling is to reduce the variance of the rare event and so achieve a narrower confidence interval than the Monte Carlo estimator, resulting in $N_{\text{IS}} \ll N_{\text{MC}}$. In general, the importance sampling distribution f' is chosen to produce the rare property more frequently, but this is not the only criterion. The optimal importance sampling distribution, denoted f^* and defined as f conditioned on the rare event, produces only traces satisfying the rare property:

$$f^* = \frac{zf}{\gamma} \quad (3)$$

This leads to the term ‘zero variance estimator’ with respect to Lz , noting that, in general, $\text{Var}(f^*) \geq 0$.

In the context of SMC f usually arises from the specifications of a model described in some relatively high level language. Such models do not, in general, explicitly specify the probabilities of individual transitions, but do so implicitly by parametrised functions over the states. We therefore consider a class of models that can be described by guarded commands [7] extended with stochastic rates. Our parametrisation is a vector of strictly positive values $\lambda \in (\mathbb{R}^+)^n$ that multiply the stochastic rates and thus maintain the absolutely continuous property between distributions. Note that this class includes both discrete and continuous time Markov chains and that in the latter case our mathematical treatment works with the embedded discrete time process.

In what follows we are therefore interested in parametrised distributions and write $f(\cdot, \lambda)$, where $\lambda = \{\lambda_1, \dots, \lambda_n\}$ is a vector of parameters, and distinguish different density functions by their parameters. In particular, μ is the original vector of the model and $f(\cdot, \mu)$ is therefore the original density. We can thus rewrite Equation (2) as

$$\gamma = \int_{\Omega} L(\omega) z(\omega) f(\omega, \lambda) d\omega$$

where $L(\omega) = f(\omega, \mu)/f(\omega, \lambda)$. We can also rewrite Equation (3)

$$f^* = \frac{zf(\cdot, \mu)}{\gamma}$$

and write for the optimal parametrised density $f(\cdot, \lambda^*)$. We define the optimum parametrised density function as the density that minimises the *cross-entropy* [16] between $f(\cdot, \lambda)$ and f^* for a given parametrisation and note that, in general, $f^* \neq f(\cdot, \lambda^*)$.

2.2 The Cross-Entropy Method

Cross-entropy [16] (alternatively *relative entropy* or Kullback-Leibler divergence) has been shown to be a uniquely correct directed measure of distance between distributions [25]. With regard to the present context, it has also been shown to be useful in finding optimum distributions for importance sampling [22][6][19].

Given two probability density functions f and f' over the same probability space Ω , the cross-entropy from f to f' is given by

$$\begin{aligned} \text{CE}(f, f') &= \int_{\Omega} f(\omega) \log \frac{f(\omega)}{f'(\omega)} d\omega = \int_{\Omega} f(\omega) \log f(\omega) - f(\omega) \log f'(\omega) d\omega \\ &= H(f) - \int_{\Omega} f(\omega) \log f'(\omega) d\omega \end{aligned} \quad (4)$$

where $H(f)$ is the entropy of f . To find λ^* we minimise $\text{CE}\left(\frac{z(\omega)f(\omega,\mu)}{\gamma}, f(\omega, \lambda)\right)$, noting that $H(f(\omega, \mu))$ is independent of λ :

$$\lambda^* = \arg \max_{\lambda} \int_{\Omega} z(\omega) f(\omega, \mu) \log f(\omega, \lambda) d\omega \quad (5)$$

Estimating λ^* directly using Equation (5) is hard, so we re-write it using importance sampling density $f(\cdot, \lambda')$ and likelihood ratio function $L(\omega) = f(\omega, \mu)/f(\omega, \lambda')$:

$$\lambda^* = \arg \max_{\lambda} \int_{\Omega} z(\omega) L(\omega) f(\omega, \lambda') \log f(\omega, \lambda) d\omega \quad (6)$$

Using Equation (6) we can construct an unbiased importance sampling estimator of λ^* and use it as the basis of an iterative process to obtain successively better estimates:

$$\tilde{\lambda}^* = \lambda^{(j+1)} = \arg \max_{\lambda} \sum_{i=1}^{N_j} z(\omega_i^{(j)}) L^{(j)}(\omega_i^{(j)}) \log f(\omega_i^{(j)}, \lambda) \quad (7)$$

N_j is the number of simulation runs on the j^{th} iteration, $\lambda^{(j)}$ is the j^{th} set of estimated parameters, $L^{(j)}(\omega) = f(\omega, \mu)/f(\omega, \lambda^{(j)})$ is the j^{th} likelihood ratio function, $\omega_i^{(j)}$ is the i^{th} path generated using $f(\cdot, \lambda^{(j)})$ and $f(\omega_i^{(j)}, \lambda)$ is the probability of path $\omega_i^{(j)}$ under the distribution $f(\cdot, \lambda^{(j)})$.

3 A Parametrised Cross-Entropy Algorithm

We consider a system of n guarded commands with vector of rate functions $\eta = (\eta_1, \dots, \eta_n)$ and corresponding vector of parameters $\lambda = (\lambda_1, \dots, \lambda_n)$. We thus define n classes of transitions. In any given state x , the probability that command $k \in \{1 \dots n\}$ is chosen is given by

$$\frac{\lambda_k \eta_k(x)}{\langle \eta(x), \lambda \rangle}$$

where η is parametrised by x to emphasise its state dependence and the notation $\langle \cdot, \cdot \rangle$ denotes a scalar product. For the purposes of simulation we consider a space of finite paths $\omega \in \Omega$. Let $U_k(\omega)$ be the number of transitions of type k occurring in ω . We therefore have

$$f(\omega, \lambda) = \prod_k^n \left((\lambda_k)^{U_k(\omega)} \prod_{s=1}^{U_k(\omega)} \frac{\eta_k(x_s)}{\langle \eta(x_s), \lambda \rangle} \right)$$

The likelihood ratios are thus of the form

$$L^{(j)}(\omega) = \prod_k^n \left(\left(\frac{\mu_k}{\lambda_k^{(j)}} \right)^{U_k(\omega)} \prod_{s=1}^{U_k(\omega)} \frac{\langle \eta(x_s), \lambda^{(j)} \rangle}{\langle \eta(x_s), \mu \rangle} \right)$$

We substitute these expressions in the cross-entropy estimator Equation (7) and for compactness substitute $z_i = z(\omega_i)$, $u_i(k) = U_k(\omega_i)$ and $l_i = L^{(j)}(\omega_i)$ to get

$$\begin{aligned} & \arg \max_{\lambda} \sum_{i=1}^N l_i z_i \log \prod_k^n \left(\lambda_k^{u_i(k)} \prod_{s=1}^{u_i(k)} \frac{\eta_k^i(x_s)}{\langle \eta^i(x_s), \lambda \rangle} \right) \\ &= \arg \max_{\lambda} \sum_{i=1}^N \sum_k l_i z_i u_i(k) \left(\log(\lambda_k) + \sum_{s=1}^{u_i(k)} \log(\eta_k^i(x_s)) - \sum_{s=1}^{u_i(k)} \log(\langle \eta^i(x_s), \lambda \rangle) \right) \end{aligned} \quad (8)$$

We partially differentiate with respect to λ_k and get the non-linear system

$$\frac{\partial F}{\partial \lambda_k}(\lambda) = 0 \Leftrightarrow \sum_{i=1}^N l_i z_i \left(\frac{u_i(k)}{\lambda_k} - \sum_{s=1}^{|\omega_i|} \frac{\eta_k^i(x_s)}{\langle \eta^i(x_s), \lambda \rangle} \right) = 0 \quad (9)$$

where $|\omega_i|$ is the length of the path ω_i .

Theorem 1. *A solution of Equation (9) is almost surely a maximum, up to a normalising scalar.*

Proof. Using a standard result, it is sufficient to show that the Hessian matrix in λ is negative semi-definite. Consider f_i :

$$f_i(\lambda) = \sum_k u_i(k) \left(\log(\lambda_k) + \sum_{s=1}^{u_i(k)} \log(\eta_k^i(x_s)) - \sum_{s=1}^{u_i(k)} \log(\langle \eta^i(x_s), \lambda \rangle) \right)$$

The Hessian matrix in λ is of the following form with $v_k^{(s)} = \frac{\eta_k(x_s)}{\langle \eta(x_s), \lambda \rangle}$ and $v_k = (v_k^{(s)})_{1 \leq s \leq U_k(\omega)}$:

$$H_i = G - D$$

where $G = (g_{kk'})_{1 \leq k, k' \leq n}$ is the following Gram matrix

$$g_{kk'} = \langle v_k, v_{k'} \rangle$$

and D is a diagonal matrix such that

$$d_{kk} = \frac{u_k}{\lambda_k^2}.$$

Note that asymptotically $d_{kk} = \frac{1}{\lambda_k} \sum_{s=1}^N v_k^{(s)}$. We write $\mathbb{1}_N = (1, \dots, 1)$ for the vector of N elements 1, hence

$$d_{kk} = \frac{1}{\lambda_k} \langle v_k, \mathbb{1}_N \rangle.$$

Furthermore, $\forall s, \sum_{k=1}^n \lambda_k v_k^{(s)} = 1$. So, $\sum_{k'=1}^n \lambda_{k'} v_{k'} = \mathbb{1}_N$. Finally,

$$d_{kk} = \sum_{k'=1}^n \frac{\lambda_{k'}}{\lambda_k} \langle v_k, v_{k'} \rangle.$$

Let $x \in \mathbb{R}^n$. To prove the theorem we need to show that $-x^t H x \geq 0$.

$$\begin{aligned} -x^t H x &= x^t D x - x^t G x \\ &= \sum_{k, k'} \frac{\lambda_{k'}}{\lambda_k} \langle v_k, v_{k'} \rangle x_k^2 - \sum_{k, k'} \langle v_k, v_{k'} \rangle x_k x_{k'} \\ &= \sum_{k < k'} \left(\left[\frac{\lambda_{k'}}{\lambda_k} x_k^2 + \frac{\lambda_k}{\lambda_{k'}} x_{k'}^2 - 2x_k x_{k'} \right] \langle v_k, v_{k'} \rangle \right) \\ &= \sum_{k < k'} \left(\sqrt{\frac{\lambda_{k'}}{\lambda_k}} x_k - \sqrt{\frac{\lambda_k}{\lambda_{k'}}} x_{k'} \right)^2 \langle v_k, v_{k'} \rangle \\ &\geq 0 \end{aligned} \tag{10}$$

The Hessian matrix H of f is of the general form

$$H = \sum_{i=1}^N l_i z_i H_i$$

which is a positively weighted sum of non-positive matrices. \square

The Hessian is negative *semi*-definite because if λ is a solution then $x\lambda, x \in \mathbb{R}^+$, is also a solution. The fact that there is a unique optimum, however, makes it conceivable to find λ^* using standard optimising techniques such as Newton and quasi-Newton methods. To do so would require introducing a suitable normalising constraint in order to force the Hessian to be negative definite. In the case of the cross-entropy algorithm of [19], this constraint is inherent because it works at the level of individual transition probabilities that sum to 1 in each state. We note here that in the case that our parameters apply to individual transitions, such that one parameter corresponds to exactly one transition, Equation (12) may be transformed to Equation (9) of [19] by constraining $\langle K, \lambda \rangle = 1$. Equation (9) of [19] has been shown in [20] to converge to f^* , implying that under these circumstances $f(\cdot, \lambda^*) = f^*$ and that it may be possible to improve our parametrised importance sampling distribution by increasing the number of parameters.

3.1 The Algorithm

Equation (9) leads to the following expression for λ_k :

$$\lambda_k = \frac{\sum_{i=1}^N l_i z_i u_i(k)}{\sum_{i=1}^N l_i z_i \sum_{s=1}^{|\omega_i|} \frac{\eta_k^i(x_s)}{\langle \eta^i(x_s), \lambda \rangle}} \quad (11)$$

In this form the expression is not useful because the right hand side is dependent on λ_k in the scalar product. Hence, in contrast to update formulae based on unbiased estimators, as given by Equation (7) and in [19, 6], we construct an iterative process based on a biased estimator but having a fixed point that is the optimum:

$$\lambda_k^{(j+1)} = \frac{\sum_{i=1}^{N_j} l_i z_i u_i(k)}{\sum_{i=1}^{N_j} l_i z_i \sum_{s=1}^{|\omega_i|} \frac{\eta_k^i(x_s)}{\langle \eta^i(x_s), \lambda \rangle}} \quad (12)$$

Equation (12) can be seen as an implementation of Equation (II) which uses the previous estimate of λ in the scalar product, however it works by reducing the distance between successive distributions, rather than by explicitly reducing the distance from the optimum. To show that the algorithm works, we first recall that Theorem I proves that there is a unique optimum (λ^*) of Equation (9) which is therefore the unique solution of Equation (II). By inspection and comparison with Equation (II), we see that any fixed point of Equation (12) is also a solution of Equation (II). Since Equation (II) has a unique solution, Equation (12) has a unique fixed point that is the optimum.

Initial Distribution. The algorithm requires an initial simulation distribution ($f(\cdot, \lambda^{(0)})$) that produces at least a few traces that satisfy the property ('successful' traces) within N_0 simulation runs. Finding $f(\cdot, \lambda^{(0)})$ for an arbitrary model may seem to be an equivalently difficult problem to estimating γ , but this is not in general the case: the rareness of the property in trace space does not imply that good parameters are rare in parameter space. In particular, when a property (e.g., failure of the system) is semantically linked to an explicit feature of the model (e.g, a command for component failure), good initial parameters can be found relatively easily by heuristic methods such as failure biasing [24]. The choice of $\lambda^{(0)}$ and N_0 is dependent on the model and the rarity of the property, but when the number of parameters is small and the property is very rare, an effective strategy is to simulate with random parameters until a suitable trace is observed. Alternatively, if the model and property are similar to a previous combination for which parameters were found, those parameters are likely to provide a good initial estimate. Increasing the parameters associated to obviously small rates may help (along the lines of failure biasing), however the rareness of a property expressed in temporal logic may not always be related to low transition probabilities. The reliability of finding good initial distributions for arbitrary systems and temporal properties is the subject of ongoing work.

Smoothing. It is conceivable that certain guarded commands play no part in traces that satisfy the property, in which case Equation (12) would make the corresponding parameter zero with no adverse effects. It is also conceivable that an important command is not seen on a particular iteration, but making its parameter zero would prevent it being seen on any subsequent iteration. To avoid this it is necessary to adopt a ‘smoothing’ strategy [19] that reduces the significance of an unseen command without setting it to zero. Smoothing therefore acts to preserve important but as yet unseen parameters. It is of particular importance when the parametrisation is close to the level of individual transition probabilities, since only a tiny fraction of possible transitions are usually seen on an individual simulation run. Typical strategies include adding a small fraction of the initial or previous parameters to every new parameter estimate. We have found that our parametrisation is often insensitive to smoothing strategy since each parameter typically governs many transitions and a large fraction of parameters are touched by each run. The smoothing strategy adopted for the examples shown below was to divide the parameter of unseen commands by two (a compromise between speed of convergence and safety). The effects of this can be seen clearly in Figure 6. Whatever the strategy, since the parameters are unconstrained it is advisable to normalise them after each iteration (i.e., $\sum_k \lambda_k = \text{const.}$) in order to judge progress.

Convergence. Given a sufficient number of successful traces from the first iteration, Equation (12) should provide a better set of parameters. In practice we have found that a single successful trace is sufficient to initiate convergence. This is in part due to the existence of a unique optimum and partly to the fact that each parameter generally governs a large number of semantically-linked transitions. The expected behaviour is that on successive iterations the number of traces that satisfy the property increases, however it is important to note that the algorithm optimises the *quality* of the distribution and that the number of traces that satisfy the property is merely emergent of that. As has been noted, in general $f(\cdot, \lambda^*) \neq f^*$, hence it is likely that fewer than 100% of traces will satisfy the property when simulating under $f(\cdot, \lambda^*)$. One consequence of this is that an initial set of parameters may produce more traces that satisfy the property than the final set (see, e.g., Figure 4).

Once the parameters have converged it is then possible to perform a final set of simulations to estimate the probability of the rare property. The usual assumption is that $N_j \ll N_{\text{IS}} \ll N_{\text{MC}}$, however it is often the case that parameters converge fast, so it is expedient to use some of the simulation runs generated during the course of the optimisation as part of the final estimation.

4 Examples

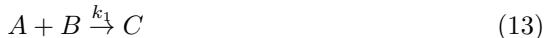
The following examples are included to illustrate the performance of our algorithm and parametrisation. The first is an example of a chemical system, often used to motivate stochastic simulation, while the second is a standard repair

model. In both cases, initial distributions were found by the heuristic of performing single simulations using parameters drawn from a Dirichlet distribution (i.e., drawn uniformly from parameter space) and using the first set of parameters that produce a path satisfying the property. For the chosen examples fewer than 500 attempts were necessary; a value less than N_j and considerably less than $1/\gamma$, the expected number of simulations necessary to see a single successful trace. All simulations were performed using our statistical model checking platform, PLASMA [13].

4.1 Chemical Network

Following the success of the human genome project, with vast repositories of biological pathway data available online, there is an increasing expectation that formal methods can be applied to biological systems. The network of chemical reactions given below is abstract but typical of biochemical systems and demonstrates the potential of SMC to handle the enormous state spaces of biological models. In particular, we demonstrate the efficacy of our algorithm by applying it to quantify two rare dynamical properties of the system.

We consider a well stirred chemically reacting system comprising five reactants (molecules of type A, B, C, D, E), a dimerisation reaction (13) and two decay reactions (14)(15):



Under the assumption that the molecules move randomly and that elastic collisions significantly outnumber unreactive, inelastic collisions, the system may be simulated using mass action kinetics as a continuous time Markov chain [8]. The semantics of Equation (13) is that if a molecule of type A encounters a molecule of type B they will combine to form a molecule of type C after a delay drawn from an exponential distribution with mean k_1 . The decay reactions have the semantics that a molecule of type C (D) spontaneously decays to a molecule of type D (E) after a delay drawn from an exponential distribution with mean k_2 (k_3). The reactions (13)(14)(15) are modelled by three guarded commands having importance sampling parameters λ_1, λ_2 and λ_3 , respectively. A typical simulation run is illustrated in Figure 1, where the x-axis is steps rather than time to aid clarity. A and B combine rapidly to form C which peaks before decaying slowly to D . The production of D also peaks while E rises monotonically.

With an initial vector of molecules $(1000, 1000, 0, 0, 0)$, corresponding to types (A, B, C, D, E) , the state space contains $\sim 10^{15}$ states. We know from a static analysis of the reactions that it is possible for the numbers of molecules of C and D to reach the initial number of A and B molecules (i.e., 1000) and that this is unlikely. To find out exactly how unlikely we consider the probabilities of the following rare properties defined in linear temporal logic: (i) $\Diamond C \geq x, x \in$

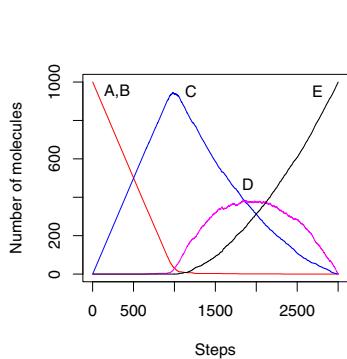


Fig. 1. A typical stochastic simulation trace of reactions (B15)

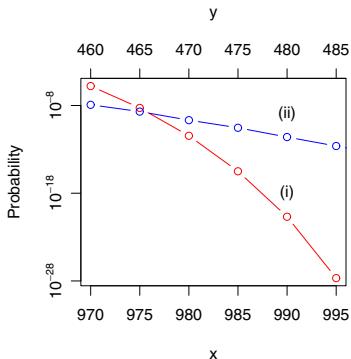


Fig. 2. (i) $\Pr[\Diamond C \geq x]$ (ii) $\Pr[\Diamond D \geq y]$

$\{970, 975, 980, 985, 990, 995\}$ and (ii) $\Diamond D \geq y, y \in \{460, 465, 470, 475, 480, 485\}$. The results are plotted in Figure 2

Having found an initial set of parameters by the heuristic means described above, the algorithm (Equation (12)) was iterated 20 times using $N_j = 1000$. Despite the large state space, this value of N_j was found to be sufficient to produce reliable results. The convergence of parameters for the property $\Diamond D \geq 470$ can be seen in Figure 3. Figure 4 illustrates that the number of paths satisfying a property can actually decrease as the quality of the distribution improves. Figure 5 illustrates the convergence of the estimate and sample variance using the importance sampling parameters generated during the course of running the algorithm. The initial set of parameters appear to give a very low variance, however this is clearly erroneous with respect to subsequent values. Noting that the variance of standard Monte Carlo simulation of rare events gives a variance approximately equal to the probability and assuming that the sample variance is close to the true variance, Figure 5 suggests that we have made a variance reduction of approximately 10^7 .

4.2 Repair Model

To a large extent the need to certify system reliability motivates the use of formal methods and thus reliability models are studied extensively in the literature. The following example is taken from [19] and features a moderately large state space of 40,320 states that can be investigated using numerical methods to corroborate our results.

The system is modelled as a continuous time Markov chain and comprises six types of subsystems $(1, \dots, 6)$ containing, respectively, $(5, 4, 6, 3, 7, 5)$ components that may fail independently. The system's evolution begins with no failures and with various probabilistic rates the components fail and are repaired. The failure rates are $(2.5\epsilon, \epsilon, 5\epsilon, 3\epsilon, \epsilon, 5\epsilon)$, $\epsilon = 0.001$, and the repair rates are $(1.0, 1.5, 1.0, 2.0, 1.0, 1.5)$, respectively. Each subsystem type is modelled by two guarded commands: one for failure and one for repair. The property under

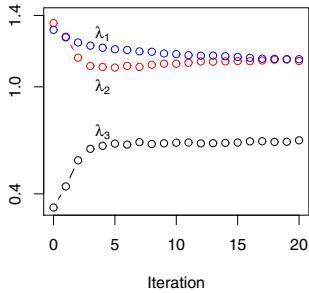


Fig. 3. Convergence of parameters for $\Diamond D \geq 470$ in the chemical model using $N_j = 1000$

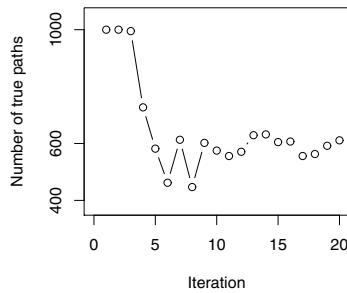


Fig. 4. Convergence of number of paths satisfying $\Diamond D \geq 470$ in the chemical model using $N_j = 1000$

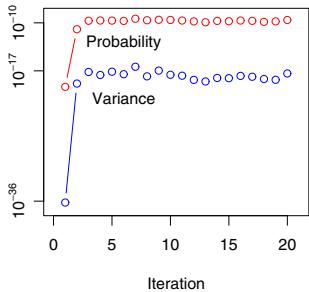


Fig. 5. Convergence of probability and sample variance for $\Diamond D \geq 470$ in the chemical model using $N_j = 1000$

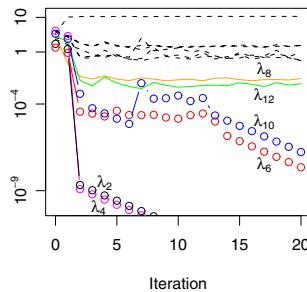


Fig. 6. Convergence of parameters and effect of smoothing (circles) in repair model using $N_j = 10000$

investigation is the probability of a complete failure of a subsystem (i.e., the failure of all components of one type), given an initial condition of no failures. This can be expressed in temporal logic as $\text{Pr}[X(\neg \text{init} \cup \text{failure})]$.

Figure 6 shows the convergence of parameters (dashed/solid lines) and highlights the effects of the adopted smoothing strategy (circles). Parameters λ_2 and λ_4 (the parameters for repair commands of types 1 and 2, respectively) are attenuated from the outset by the convergence of the other parameters (because of the normalisation). Once their values are small relative to the normalisation constant (12 in this case), their corresponding commands no longer occur and their values experience exponential decay as a result of smoothing (division by two at every subsequent step). Parameters λ_6 and λ_{10} (the parameters for repair commands of types 3 and 5, respectively) converge for 12 steps but then also decay. The parameters for the repair commands of types 4 and 6 (solid lines) are the smallest of the parameters that converge. The fact that the repair transitions are made less likely by the algorithm agrees with the intuition that we are interested in direct paths to failure. The fact that they are not necessarily made zero reinforces the point that the algorithm seeks to consider *all* paths to failure, including those that have intermediate repairs.

Figure 7 plots the number of paths satisfying $X(\neg \text{init} \cup \text{failure})$ and suggests that for this model the parametrised distribution is close to the optimum. Figure 8 plots the estimated probability and sample variance during the course of the algorithm and superimposes the true probability calculated by PRISM [26]. The long term average agrees well with the true value (an error of -1.7%, based on an average excluding the first two estimates), justifying our use of the sample variance as an indication of the efficacy of the algorithm: our importance sampling parameters provide a variance reduction of more than 10^5 .

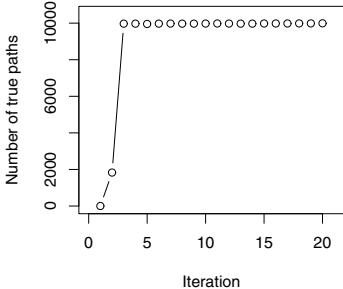


Fig. 7. Convergence of number of paths satisfying $X(\neg \text{init} \cup \text{failure})$ in the repair model using $N_j = 10000$

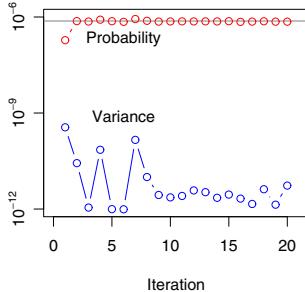


Fig. 8. Convergence of estimated probability and sample variance for repair model using $N_j = 10000$. True probability shown as horizontal line

5 Conclusions and Future Work

Statistical model checking addresses the state space explosion associated with exact probabilistic model checking by estimating the parameters of an empirical distribution of executions of a system. By constructing an executable model, rather than an explicit representation of the state space, SMC is able to quantify and verify the performance of systems that are intractable to an exhaustive approach. SMC trades certainty for tractability and often offers the only feasible means to certify real-world systems. Rare properties pose a particular problem to Monte Carlo simulation methods because the properties are difficult to observe and the error in their estimated probabilities is difficult to bound. Importance sampling is a well-established means to reduce the variance of rare events but requires the construction of a suitable importance sampling distribution without resorting to the exploration of the entire state space.

We have devised a natural parametrisation for importance sampling and have provided a simple algorithm, based on cross-entropy minimisation, to optimise the parameters for use in statistical model checking. We have shown that our parametrisation leads to a unique optimum and have demonstrated that with very few parameters our algorithm can make significant improvements in the efficiency of statistical model checking. We have shown that our approach is applicable to standard reliability models and to the kind of huge state space

models found in systems biology. We therefore anticipate that our methodology has the potential to be applied to many complex natural and man-made systems.

An ongoing challenge is to find ways to accurately bound the error of results obtained by importance sampling. Specifically, the sample variance of the results may be a very poor indicator of the true variance (i.e. with respect to the unknown true probability). Recent work has addressed this problem using Markov chain coupling applied to a restricted class of models and logic I^{C} , but a simple universal solution remains elusive. A related challenge is to find precise means to judge the quality of the importance sampling distributions we create. Our algorithm finds an optimum based on an automatic parametrisation of a model described in terms of guarded commands. Linking the importance sampling parametrisation to the description of the model in this way gives our approach an advantage when the rare property is related to semantic features expressed in the syntax. Potentially confounding this advantage is the fact that the syntactic description is likely optimised for compactness or convenience, rather than consideration of importance sampling. As a result, there may be alternative ways of describing the same model that produce better importance sampling distributions. Applying existing work on the robustness of estimators, we hope to adapt our algorithm to provide hints about improved parametrisation.

References

1. Barbot, B., Haddad, S., Picaronny, C.: Coupling and Importance Sampling for Statistical Model Checking. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 331–346. Springer, Heidelberg (2012)
2. 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 2010. LNCS, vol. 6117, pp. 32–46. Springer, Heidelberg (2010)
3. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: Uppaal — a Tool Suite for Automatic Verification of Real-Time Systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
4. Chernoff, H.: A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations. *Ann. Math. Statist.* 23(4), 493–507 (1952)
5. Clarke, E.M., Zuliani, P.: Statistical Model Checking for Cyber-Physical Systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 1–12. Springer, Heidelberg (2011)
6. De Boer, P.-T., Nicola, V.F., Rubinstein, R.Y.: Adaptive importance sampling simulation of queueing networks. In: Winter Simulation Conference, vol. 1, pp. 646–655 (2000)
7. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 453–457 (1975)
8. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry* 81, 2340–2361 (1977)
9. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: NDSS (2008)

10. Heidelberger, P.: Fast simulation of rare events in queueing and reliability models. *ACM Trans. Model. Comput. Simul.* 5, 43–85 (1995)
11. 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)
12. Hoeffding, W.: Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association* 58(301), 13–30 (1963)
13. 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)
14. 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)
15. Kahn, H.: Stochastic (Monte Carlo) Attenuation Analysis. Technical Report P-88, Rand Corporation (July 1949)
16. Kullback, S.: Information Theory and Statistics. Dover (1968)
17. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
18. Metropolis, N., Ulam, S.: The Monte Carlo Method. *Journal of the American Statistical Association* 44(247), 335–341 (1949)
19. Ridder, A.: Importance sampling simulations of markovian reliability systems using cross-entropy. *Annals of Operations Research* 134, 119–136 (2005)
20. Ridder, A.: Asymptotic optimality of the cross-entropy method for markov chain problems. *Procedia Computer Science* 1(1), 1571–1578 (2010)
21. Rubino, G., Tuffin, B. (eds.): Rare Event Simulation using Monte Carlo Methods. Wiley (2009)
22. Rubinstein, R.: The Cross-Entropy Method for Combinatorial and Continuous Optimization 1, 127–190 (1999)
23. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: A statistical model-checker and analyzer for probabilistic systems. In: QEST, pp. 251–252. IEEE (September 2005)
24. Shahabuddin, P.: Importance Sampling for the Simulation of Highly Reliable Markovian Systems. *Management Science* 40(3), 333–352 (1994)
25. Shore, J., Johnson, R.: Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy. *IEEE Transactions on Information Theory* 26(1), 26–37 (1980)
26. The PRISM website, <http://www.prismmodelchecker.org>
27. 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)
28. Younes, H.L.S.: Ymer: A Statistical Model Checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005)

Timed Relational Abstractions for Sampled Data Control Systems

Aditya Zutshi¹, Sriram Sankaranarayanan¹, and Ashish Tiwari^{2,*}

¹ University of Colorado, Boulder, CO

{aditya.zutshi,srirams}@colorado.edu

² SRI International, Menlo Park, CA

ashish.tiwari@sri.com

Abstract. In this paper, we define timed relational abstractions for verifying sampled data control systems. Sampled data control systems consist of a plant, modeled as a hybrid system and a synchronous controller, modeled as a discrete transition system. The controller computes control inputs and/or sends control events to the plant based on the periodically sampled state of the plant. The correctness of the system depends on the controller design as well as an appropriate choice of the controller sampling period.

Our approach constructs a timed relational abstraction of the hybrid plant by replacing the continuous plant dynamics by relations. These relations map a state of the plant to states reachable within the sampling time period. We present techniques for building timed relational abstractions, while taking care of discrete transitions that can be taken by the plant between samples. The resulting abstractions are better suited for the verification of sampled data control systems. The abstractions focus on the states that can be observed by the controller at the sample times, while abstracting away behaviors between sample times conservatively. The resulting abstractions are discrete, infinite-state transition systems. Thus conventional verification tools can be used to verify safety properties of these abstractions. We use k-induction to prove safety properties and bounded model checking (BMC) to find potential falsifications. We present our idea, its implementation and results on many benchmark examples.

1 Introduction

We present techniques for verifying safety properties of sampled data control systems using timed relational abstractions. Sampled data control systems consist of a discrete controller that periodically senses the state of a continuous physical plant, and actuates by setting inputs or sending control commands to the plant. Sampled data control systems are quite common in practice. Complex (network) control systems often involve many control tasks that are scheduled periodically, with each task controlling a different aspect of the plant. The cadencing of these tasks to enforce the safety and stability of

* Zutshi and Sankaranarayanan's work supported by NSF Career grant CNS-0953941. Tiwari's work supported in part by DARPA under subcontract No. VA-DSR 21806-S4 under prime contract No. FA8650-10-C-7075, and NSF grants CSR-0917398 and SHF:CSR-1017483. All opinions expressed are those of the authors and not necessarily of the US NSF or DARPA.

the system is an important problem. The choice of sampling period is crucial: a small sampling time can place infeasible constraints on the scheduling policy, whereas large sampling times can cause instabilities or safety violations.

In this paper, we consider a simple and natural model of a sampled data control system. The controller is modeled by an infinite state (linear) transition system. It communicates synchronously with a plant modeled by an affine hybrid automaton. The controller runs periodically with a fixed sampling period $T_s > 0$. At each time period, the controller senses the state of the plant and performs controller actions that may include (a) setting control input signals for the plant, and (b) “commanding” the plant to execute a controlled discrete transition, resulting in an instantaneous jump and a mode change in the plant.

Our verification approach uses the idea of timed relationalization, extending the idea of untimed relational abstractions [35]. A timed relational abstraction considers the set of states of the plant that are potentially observable by the controller at the sample times, while safely abstracting away all the intermediate states. To this end, we build relations that map a state of the plant at the beginning of a sampling period to states that can be reached at the end. Using these relations, the entire plant can be safely abstracted away by a discrete, infinite-state transition system. This system is composed together with the controller to yield an overall discrete system that can be analyzed by existing tools such as k -induction [36], bounded model-checking [5] and abstract interpretation [10][22], while exploiting advances in abstract domains, SAT and SMT solvers.

There are two key challenges in constructing the timed abstraction: (a) dealing with the continuous dynamics inside a mode, and (b) handling autonomous transitions that can be taken by the plant between two sampling periods. For systems with affine dynamics the former problem is solved by computing a matrix exponential for the matrices defining the dynamics in each mode [27]. However, the numerical exponential computation is potentially unsound due to round-off and truncation errors. Likewise, solving for autonomous transitions involves computing a symbolic matrix exponential to deal with the unknown switching times for each transition. To solve both problems, we exploit advances in interval arithmetic to compute guaranteed enclosures to the matrix exponential [28][29][18][6]. This yields interval linear relations. We then use a *template-based* mechanism using SMT solvers to abstract the resulting interval linear relations in terms of relations expressible in linear arithmetic.

We have implemented our approach to relationalization and present an extensive evaluation over a set of benchmark systems. Our evaluation performs a relational abstraction of the plant using the techniques described in this paper. The resulting abstraction is analyzed using the SAL tool-set from SRI [32][40]. The results of our evaluation are quite promising: we show that our techniques can successfully handle complex sampled data control systems efficiently and soundly. We compare our approach with the SpaceEx tool that implements symbolic model checking for affine hybrid automata using support functions [16]. Our approach compares quite favorably with the various options available in SpaceEx: providing safety proofs in many cases where SpaceEx fails to prove safety over finite time horizons. On the other hand, we note that the finite time horizon bounds on the reachable state-space established by SpaceEx can sometimes be used as strengthenings for k -induction in our approach to obtain significant

speedups. Our implementation, the data from our experiments along with an extended version of the paper will be available online¹

Motivating Examples: We discuss two simple motivating examples that clearly illustrate the need for verification of sampled data control systems.

Consider a proportional-integral (PI) controller defined by $u' := -30x - y$ composed with a plant defined by $\dot{x} = 5x + u$, $\dot{y} = x$. With a period $T_s = 0.1s$, the controller is able to stabilize the plant but fails to do so with a period $T_s = 0.5s$.

Consider an inverted pendulum controller:

$$u' := \begin{cases} -16, & y \geq 2 \vee 16x - y \leq -10 \\ 16, & y \leq -2 \vee 16x - y \geq 10 \\ u, & \text{otherwise} \end{cases} .$$

The linearized plant has the dynamics $\dot{x} = y$ $\dot{y} = 20x + 16y + 4u$. If the controller is implemented in the continuous domain, it results in a stable system. However, a digital implementation, regardless of the sampling period, is unable to stabilize the pendulum.

We now discuss the related work.

Relational Abstractions: Relational abstractions have been used primarily for checking liveness properties [4][30]. There are many subtle distinctions between the various forms of relational abstractions used. Transition invariants [30], used in termination proofs, relate the current state to *any* previous state at a given program location. Likewise, progress invariants relate the current state and the *immediately* previous state at a given location [20]. Podelski and Wagner provide a verification procedure for (region) stability properties of hybrid systems [31], wherein they derive *binary reachability relations* over trajectories of a hybrid system, similar in spirit to a relational abstraction. Note that Podelski and Wagner use a hybrid system reachability tool to compute their abstractions in the first place. The techniques in this paper and our previous work [33] are meant to solve the reachability problem using these relations.

Our previous work explored the idea of abstracting the dynamics inside each discrete mode of a hybrid automaton by an *untimed relational abstraction* [35]. The relational abstractions presented here capture the relationship between the current state at time $t = t_0$ and any state reachable at time $t = t_0 + T_s$ units. The consideration of the sampling time T_s is essential for verifying sampled data control systems. Furthermore, it is also important to note that unlike untimed relational abstraction, it is essential for the abstraction presented here to account for the plant's discrete transitions that can be taken in the time interval $t \in [t_0, t_0 + T_s]$.

Abstractions of Hybrid Systems: *Discrete abstractions* have been widely studied and applied for verifying safety properties of hybrid systems [1][39]. The use of counterexample-guided abstraction refinement has also been investigated in the past [18]. In this paper, the proposed abstraction yields a discrete but infinite state system. Hybridization is a technique for converting nonlinear systems into affine hybrid systems by subdividing the invariant region into numerous subregions and approximating

¹ <http://systems.cs.colorado.edu/research/cyberphysical/relational>

the dynamics as a hybrid system by means of a linear differential inclusion in each region [23][31]. However, such a subdivision is expensive as the number of dimensions increases and often infeasible if the invariant region is unbounded.

Flowpipe Construction: Reasoning about the reachable set of states for flows of affine hybrid systems through flowpipe approximation has long been dominant approach for checking safety properties [37], using various representations for sets of states including ellipsoids [25], zonotopes [17], template polyhedra [33], and support functions [19]. The tool SpaceEx implements numerous improvements to these techniques with impressive performance on some benchmarks with a large number of system variables [16].

Synchronous Systems: Techniques for verifying synchronous system models, with piecewise constant continuous dynamics, have been studied in the past, notably by Halbwachs et al. [22] and as part of the NBAC tool by Jeannet et al. [24]. Our work considers a synchronous controller with affine hybrid plants. Furthermore, we consider the idea of an up front relationalization of the plant dynamics, enabling a verification procedure to focus purely on discrete systems.

2 Sampled Data Control Systems

Let \mathbb{R} denote the set of real numbers. We use a, \dots, z with subscripts to denote (column) vectors and A, \dots, Z to denote matrices. For a $m \times n$ matrix A , the row vector A_i , for $1 \leq i \leq m$, denotes the i^{th} row.

We discuss models for sampled data control systems. Figure 1 shows the schematic diagram for such a control system consisting of a discrete controller communicating with a hybrid plant. The controller has a time period $T_s > 0$. Every T_s time units, the controller “senses” the state of the hybrid plant and synchronizes to change the mode of the plant. The commands can take the form of (a) events enabling a discrete transition of the plant, or (b) values for control inputs that are assumed to be held constant throughout the sample time period. We model the controller as a discrete transition system [20].

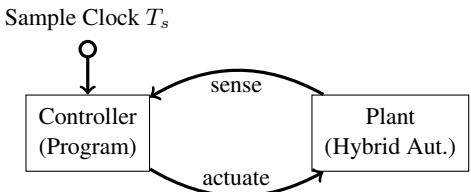


Fig. 1. Schematic for a Sampled Data Control System

Definition 1 (Discrete Transition System). A discrete transition system Π is a tuple $\langle L, \mathbf{x}, \mathcal{T}, \ell_0, \Theta \rangle$ wherein, L is a finite set of discrete locations; $\mathbf{x} : (x_1, \dots, x_n)$ is a set of variables with variable x_i of type $\text{type}(x_i)$; \mathcal{T} is a set of discrete transitions; $\ell_0 \in L$ is the initial location; and $\Theta[\mathbf{x}]$ is an assertion capturing the initial values for \mathbf{x} .

Each transition $\tau \in \mathcal{T}$ is of the form $\langle \ell, m, \rho_\tau \rangle$, wherein ℓ is the pre-location of the transition and m is the post-location. The relation $\rho_\tau[\mathbf{x}, \mathbf{x}']$ represents the transition relation over current state variables \mathbf{x} and next state variables \mathbf{x}' .

We now discuss the overall model for the plant as a hybrid automaton with controlled and uncontrolled transitions.

Definition 2 (Plant Model). A plant \mathcal{P} is an extended hybrid automaton described by a tuple $\langle \mathbf{x}, \mathbf{u}, Q, \mathcal{F}, \mathcal{X}, \mathcal{T}, q_0, X_0 \rangle$, wherein,

- $\mathbf{x} : (x_1, \dots, x_n)$ denotes the continuous state variables, and $\mathbf{u} : (u_1, \dots, u_m)$ the control inputs,
- Q is a finite set of discrete modes. $q_0 \in Q$ is the initial mode and X_0 the initial set of states.
- \mathcal{F} maps each discrete mode $q \in Q$ to an ODE $\frac{dx}{dt} = F_q(\mathbf{x}, \mathbf{u}, t)$.
- \mathcal{X} maps each discrete mode $q \in Q$ to a mode invariant $\mathcal{X}(q) \subseteq \mathbb{R}^n$.
- \mathcal{T} represents a set of discrete transitions. Each transition $\tau \in \mathcal{T}$ is a tuple $\langle s, t, \gamma, U \rangle$ wherein s, t represent the pre- and post- mode respectively. $\gamma[\mathbf{x}]$ is the transition guard assertion, and U maps each variable $x_i \in \mathbf{x}$ to an update function $U_i(\mathbf{x})$. The transition relation for τ is defined as $\rho_\tau(\mathbf{x}, \mathbf{x}') : \gamma(\mathbf{x}) \wedge \mathbf{x}' = U(\mathbf{x})$.
- We partition the transitions in \mathcal{T} as autonomous transitions \mathcal{T}_{aut} and controlled transitions \mathcal{T}_{ctrl} . Autonomous transitions can be taken by the plant non-deterministically, whenever enabled. On the other hand, controlled transitions are taken upon an explicit command by the controller.

The state of the plant is a tuple $(q, \mathbf{x}, \mathbf{u})$ consisting of the current mode q , state values \mathbf{x} and controller input \mathbf{u} . Note that the control input is set at the beginning of a time period, and is assumed to remain constant throughout the period.

The overall sampled data control system is a tuple $\langle \mathcal{C}, \mathcal{P}, \mu, T_s \rangle$ of a discrete controller transition system \mathcal{C} , a hybrid plant model \mathcal{P} and a mapping μ from variables in \mathcal{C} to control inputs \mathbf{u} of \mathcal{P} . A given sampling time $T_s > 0$ specifies the periodicity of the controller execution. The state of the system is represented by the joint state of the plant σ_P and σ_C of the controller. We assume that the computations of the controller take zero (or negligible time) compared to the sampling period. Furthermore, we assume that the commands issued by the controller are in the form of an input \mathbf{u} for the next time period, and/or a command to execute a discrete transition. Finally, to avoid considering improbable “race conditions”, we assume that the plant itself may not execute autonomous discrete transitions at sample time instances when the controller executes². The overall system evolves in one of two ways:

1. At sample times $t = nT_s$ for $n \in \mathbb{Z}$, a controlled transition is taken based on the current state of the plant and the controller. The transition updates the controller state, the values of the plant inputs and can also command the plant to execute a discrete transition out of its current mode.
2. Between two sample times $t \in (nT_s, (n+1)T_s)$, the state of the plant evolves according to its current mode q , continuous variables \mathbf{x} and input \mathbf{u} . If an autonomous transition $\tau \in \mathcal{T}_{aut}$ is enabled, then it may be non-deterministically executed by the plant, possibly changing the plant’s state instantaneously.

² This assumption can be relaxed to allow such simultaneous executions, provided the plant and the controller do not attempt to update the same state variable.

A plant is *affine* iff (a) for each discrete mode q , the dynamics are of the form $\frac{dx}{dt} = A_q \mathbf{x} + B_q \mathbf{u} + b_q$, (b) the initial condition Θ and guards γ_τ for each transition τ , are linear arithmetic formulae, and (c) the update functions U_τ are affine.

Why Autonomous Transitions? The ability to model autonomous transitions is quite important in practice. Real life plants are often multi modal with mode changes that can be effected by exogenous user inputs, disturbance inputs, system failures and other exceptional situations. Examples include pump failures or occlusion events observed in models of drug administration using infusion pumps [2][34], failsafe models of spacecraft control systems, wherein exogenous disturbance inputs can cause mode changes [7]. Another reason for autonomous transitions includes the modeling of actuation delays. Autonomous transitions can be used to model delays between controller commands and their actuation in networked control systems.

3 Relationalization

In this section, we discuss the notion of timed relationalizations for plants in a sampled data control system. The basic idea behind relationalization is to build a relation $R_P(q', \mathbf{x}', q, \mathbf{x}, \mathbf{u})$ of all possible pairs of states (q', \mathbf{x}') and (q, \mathbf{x}) such that (a) the plant is in the state (q, \mathbf{x}) at the start time $t = t_0$, (b) it reaches the state (q', \mathbf{x}') at time $t = t_0 + T_s$, and (c) \mathbf{u} is the constant controller input for $t \in (t_0, t_0 + T_s]$. Note that the discrete modes q, q' may be different, depending on whether an autonomous transition is taken by the plant between two samplings.

Let us suppose a relation R_P can be built that can characterize all pairs (q', \mathbf{x}') that a controller can observe at the next time step, given that (q, \mathbf{x}) was observed at the current time step and \mathbf{u} was the control input. As a result, we may construct a purely discrete abstraction of the sampled data control system wherein the behavior of the plant between two samplings is entirely captured by R_P . Therefore, the resulting discrete transition system can be verified using a host of approaches for verification of discrete programs. Furthermore, since our goal is to perform safety verification, we do not need to compute the exact relation R_P , but only an over-approximation of it.

We will now describe techniques for constructing timed relational abstractions.

Example 1. Consider the hybrid plant model shown in Figure 2 with two state variables x, y and no control inputs. The matrices defining the dynamics are

$$A_1 : \begin{pmatrix} -1.5 & 1.2 \\ 1.3 & 0.2 \end{pmatrix} \quad b_1 : \begin{pmatrix} 1.0 \\ -0.5 \end{pmatrix} \quad A_2 : \begin{pmatrix} 2 & 1.2 \\ 0.1 & -3.6 \end{pmatrix} \quad b_2 : \begin{pmatrix} -0.6 \\ -0.6 \end{pmatrix}.$$

There are two modes n_0 and n_1 with an autonomous transition τ_1 from n_0 to n_1 and a controlled transition τ_2 from n_1 back to n_0 . Relationalization of this automaton will need to consider 3 cases: (a) the automaton remains entirely inside the mode n_0 during a sample interval, (b) the automaton remains entirely inside mode n_1 and (c) the automaton switches from mode n_0 to n_1 sometime during a sampling interval.

We first discuss relational abstraction for the case when the plant remains in some mode q during a sampling period without any autonomous transitions taken in between. This

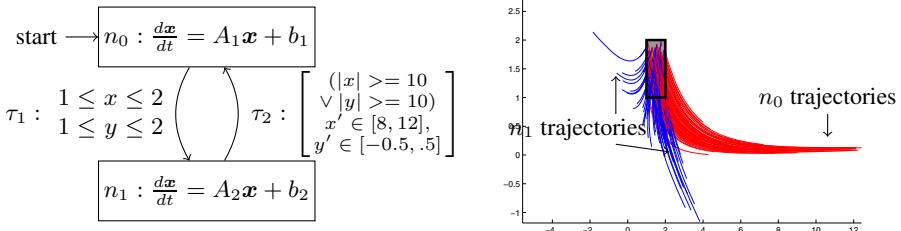


Fig. 2. A simple affine hybrid automaton with an autonomous transition τ_1 and controlled transition τ_2 . Some sample trajectories of the automaton are shown with the autonomous transition being taken. The red colored trajectories belong to mode n_0 and the blue colored trajectories to mode n_1 . The guard set is shown in thick lines.

situation is abstracted by a relation $R_q(\mathbf{x}, \mathbf{u}, \mathbf{x}')$ that relates all plant states (q, \mathbf{x}) at some time t and state (q, \mathbf{x}') at time $t + T_s$ with control input \mathbf{u} . The resulting R_q for each $q \in Q$ will form a disjunct in the overall relation R_P for the plant.

Definition 3 (Timed Relational Abstraction). Consider a continuous system specified by a time invariant ODE $\frac{d\mathbf{x}}{dt} = f(\mathbf{x}, \mathbf{u})$ for $\mathbf{x} \in X$ and control inputs $\mathbf{u} \in U$.

A relation $R(\mathbf{x}, \mathbf{u}, \mathbf{x}')$ is a timed relational abstraction with sample time T_s of the continuous system iff for all time trajectories $\mathbf{x}(t)$ of the ODE with constant control input $\mathbf{u}(t) = \mathbf{u}$, we have $(\mathbf{x}(0), \mathbf{u}, \mathbf{x}(T_s)) \in R$.

Since we assume that the dynamics are time invariant, the starting time of the observation can be arbitrarily set to $t_0 = 0$. Time varying dynamics can be treated by lifting this assumption and specifying the value of the time t as part of the state \mathbf{x} of the system.

We now consider the timed relational abstraction for a system with affine dynamics given by $\frac{d\mathbf{x}}{dt} = A\mathbf{x} + B\mathbf{u} + \mathbf{b}$. We note that the solution of the ODE can be written as $\mathbf{x}(t) = e^{tA}\mathbf{x}(0) + \int_{s=0}^t e^{(t-s)A}(B\mathbf{u}(s) + \mathbf{b}) ds$. If the matrix A is invertible and $\mathbf{u}(s) = \mathbf{u}$ for $s \in [0, T_s]$, we may write the resulting relation as

$$\mathbf{x}(T_s) = e^{T_s A} \mathbf{x}(0) + A^{-1}(e^{T_s A} - I)(B\mathbf{u} + \mathbf{b}).$$

For general A , we write the result as

$$\mathbf{x}(T_s) = e^{T_s A} \mathbf{x}(0) + P(A, T_s)(B\mathbf{u} + \mathbf{b}), \text{ wherein } P(A, t) = \sum_{j=0}^{\infty} \frac{A^j t^{j+1}}{(j+1)!}.$$

In theory, given T_s and A , we may compute the matrices $e^{T_s A}$ and $P(A, T_s)$ to arbitrary precision. This yields an affine expression for $\mathbf{x}(T_s)$ in terms of $\mathbf{x}(0), \mathbf{u}$.

In practice, however, arbitrary precision computation of the exponential map is often impractical, unless the matrix A is known to be diagonalizable with restrictions on its eigenvalues, or nilpotent. Therefore, for a general matrix A , we resort to error-prone numerical computations of $e^{T_s A}$ and $P(A, T_s)$.

The loss of soundness can be alleviated by using sophisticated numerical approximation schemes [27]. In particular, we can estimate both matrices using interval arithmetic calculations as $e^{T_s A} \in [\underline{E}_s, \overline{E}_s]$ and $P(A, T_s) \in [\underline{P}_s, \overline{P}_s]$ by taking into account the arithmetic and truncation errors of the resulting power series expansions [6, 29, 18]. Therefore, the resulting relationalization obtained is *interval linear* of the form $\mathbf{x}' \in [\underline{E}_s, \overline{E}_s]\mathbf{x} - [\underline{P}_s, \overline{P}_s](B\mathbf{u} + \mathbf{b})$ which stands for the logical formula

$$R(\mathbf{x}, \mathbf{u}, \mathbf{x}') : (\exists E \in [\underline{E}_s, \overline{E}_s], P \in [\underline{P}_s, \overline{P}_s]) \mathbf{x}' = E\mathbf{x} - P(B\mathbf{u} + \mathbf{b}).$$

As such, the relation above cannot be expressed in linear arithmetic. We will expand upon the treatment of interval linear relations later in this section.

Example 2. Going back to the system in Ex. II, we find relational abstractions for mode n_0 when the system does not take an autonomous transition within the sampling period of 0.2 time units. Using a numerically computed matrix exponential, we obtain the relation $\mathbf{x}' = \begin{pmatrix} 0.7669 & 0.214 \\ 0.232 & 1.07 \end{pmatrix} \mathbf{x} + \begin{pmatrix} 0.1635 \\ -0.079 \end{pmatrix}$. On the other hand, using the interval arithmetic based method described by Goldsztejn [18], we obtain the relation

$$\begin{aligned} \mathbf{x}' \in & \left(\begin{array}{cc} [0.7669282852020186, 0.7669282852020187] & [0.2139643726426075, 0.2139643726426076] \\ [0.2317947337083272, 0.2317947337083273] & [1.0700444963848672, 1.0700444963848673] \end{array} \right) \mathbf{x} \\ & + \left(\begin{array}{c} [0.1635149326785402, 0.1635149326785403] \\ [-0.0789845829507958, -0.0789845829507957] \end{array} \right). \end{aligned}$$

While pathological cases for matrix exponential computation are known (Cf. Goldsztejn [18]), the rather tight interval bounds for the exponential seem to be quite common in our benchmarks, and therefore, the use of numerically computed matrix exponentials may be quite satisfactory for many applications, wherein the dynamics are obtained as an approximation of the physical reality in the first place.

Applying the same for mode n_2 , we obtain the relation $\mathbf{x}' = \begin{pmatrix} 1.5245 & 0.2181 \\ 0.0182 & 0.4885 \end{pmatrix} \mathbf{x} + \begin{pmatrix} -0.1626 \\ -0.0867 \end{pmatrix}$. Once again, an interval computation yields intervals of width 10^{-16} or less centered around the numerically computed value.

3.1 Dealing with Autonomous Transitions

Thus far, we have described a simple relationalization scheme under the assumption that no autonomous transitions were taken by the plant during a sampling time period. We will now describe the treatment of autonomous transitions that can be taken by the plant between two successive samplings. In general, there is no *a priori* bound on the number of such transitions that a plant can take in any given period $(nT_s, (n+1)T_s)$. Even if the plant is assumed to be non-Zeno, any relationalization has to capture the effects of the plant executing a finite but unbounded number of transitions. We remedy this situation by making two assumptions regarding the plant: (a) There is a minimum dwell time $T_D > 0$ for each mode q of the plant. In other words, whenever a run of the plant enters some mode q , it remains there for at least T_D time units before an

autonomous transition is enabled. (b) No autonomous transitions can be taken precisely at the time instant $t = jT_s$ for $j \in \mathbb{Z}$.

The first assumption provides a bound $N = \left\lceil \frac{T_s}{T_D} \right\rceil$ on the maximum number of autonomous transitions taken inside a sampling interval. For this paper, we will assume that $N = 1$ to simplify the presentation, i.e., the controller is assumed to sample the plant fast enough to restrict the number of autonomous transitions in any sample period to at most 1. The second assumption allows us to use the standard *interleaving semantics* when the relationalization of the plant and the system are composed. This assumption fails if the execution time of the controller is not negligible compared to the time scale of the plant dynamics, as is sometimes the case. However, if bounds are known on the execution times, we may compute relationalizations of the plant for two time steps, one for the controller step and the other for the sampling period. Likewise, the basic ideas presented here extend to more sophisticated task execution schedules for control tasks.

Let us assume that a single autonomous transition $\tau : \langle q_1, q_2, \rho \rangle$ is taken during the time $t \in (0, T_s)$. Our goal is to derive a relation $R_\tau((q_1, \mathbf{x}), \mathbf{u}, (q_2, \mathbf{x}'))$ characterizing all possible pairs of states (q_1, \mathbf{x}) and (q_2, \mathbf{x}') so that the plant may evolve from continuous state \mathbf{x} in mode q_1 at time $t = 0$ to the state (q_2, \mathbf{x}') at time $t = T_s$ with the transition τ taken at some time instant $0 < t < T_s$. The resulting R_τ for each $\tau \in \mathcal{T}_{aut}$ will form a disjunct of the overall relation R_P for the plant.

Figure 3 summarizes the situation diagrammatically. We let \mathbf{y} be the valuation to continuous variables just prior to τ being taken and \mathbf{y}' be the valuation just after τ is taken. Let t be the time instant at which τ is taken. Let the dynamics in mode q_i for $i = 1, 2$ be given by $\frac{d\mathbf{x}}{dt} = A_i \mathbf{x} + B_i \mathbf{u} + \mathbf{b}_i$. Therefore, $\mathbf{x} = e^{-tA_1} \mathbf{y} - P(A_1, -t)(B_1 \mathbf{u} + \mathbf{b}_1) \wedge \mathbf{x}' = e^{(T_s-t)A_2} \mathbf{y}' - P(A_2, T_s - t)(B_2 \mathbf{u} + \mathbf{b}_2)$. The overall relation is given by

$$R_\tau(\mathbf{x}, \mathbf{u}, \mathbf{x}') : (\exists t, \mathbf{y}, \mathbf{y}') \left(\begin{array}{l} \mathbf{x} = e^{-tA_1} \mathbf{y} - P(A_1, -t)(B_1 \mathbf{u} + \mathbf{b}_1) \\ \mathbf{x}' = e^{(T_s-t)A_2} \mathbf{y}' - P(A_2, T_s - t)(B_2 \mathbf{u} + \mathbf{b}_2) \\ 0 < t < T_s \wedge \rho_\tau(\mathbf{y}, \mathbf{y}') \end{array} \right). \quad (1)$$

Note that we have chosen to encode $\mathbf{x} = e^{-tA_1} \mathbf{y}$ instead of encoding the dynamics in the forward direction $\mathbf{y} = e^{tA_1} \mathbf{x}$. This seemingly arbitrary choice will be seen to make the subsequent quantifier elimination problem easier.

Eliminating Quantifiers: The main problem with the relation R_τ derived in Eqn. (1) is that the matrices e^{tA_i} and $P(A_i, t)$ are, in general, *transcendental functions* of time. It is computationally intractable to manipulate these relations inside decision procedures. To further complicate matters, the variable t is existentially quantified. Removing this

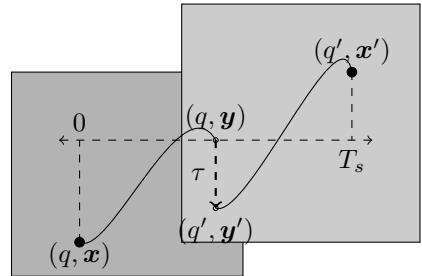


Fig. 3. Schematic for relational abstraction of a single autonomous transition

quantifier poses yet another challenge. However, our goal will be to derive an over-approximation of R_τ expressible in linear arithmetic.

To this end, the main challenge is to construct a good quality and linear over-approximation $R_\tau^a(\mathbf{x}, \mathbf{u}, \mathbf{x}')$ of the relation R_τ . We address this challenge using interval arithmetic techniques.

Interval over-approximation We subdivide the interval $[0, T_s]$ into $M > 0$ subintervals each of width $\delta = \frac{T_s}{M}$. Next, we consider each subinterval of the form $t \in [i\delta, (i+1)\delta)$ and use interval arithmetic evaluation for the functions e^{tA_i} and $P(A_i, t)$ to obtain a conservative approximation valid for the subinterval. In effect, we over-approximate R_τ as a disjunction

$$R_\tau^I : \bigvee_{0 \leq i < M} (\exists \mathbf{y}, \mathbf{y}') \left(\begin{array}{l} \mathbf{x} \in [\underline{E}_{i,1}, \overline{E}_{i,1}] \mathbf{y} - [\underline{P}_{i,1}, \overline{P}_{i,1}] (B_1 \mathbf{u} + \mathbf{b}_1) \wedge \\ \mathbf{x}' \in [\underline{E}_{i,2}, \overline{E}_{i,2}] \mathbf{y}' - [\underline{P}_{i,2}, \overline{P}_{i,2}] (B_2 \mathbf{u} + \mathbf{b}_2) \wedge \\ \rho_\tau(\mathbf{y}, \mathbf{y}') \end{array} \right), \quad (2)$$

wherein $[\underline{E}_{i,1}, \overline{E}_{i,1}]$ is a safe interval enclosure of $e^{(-(i+1)\delta, -i\delta]A_1}$ while $[\underline{E}_{i,2}, \overline{E}_{i,2}]$ is an enclosure of $e^{(T_s - [i\delta, (i+1)\delta])A_2}$. Likewise, $[\underline{P}_{i,1}, \overline{P}_{i,1}]$ and $[\underline{P}_{i,2}, \overline{P}_{i,2}]$ are safe enclosures of $P(A_1, [-(i+1)\delta, -i\delta])$ and $P(A_2, (T_s - [i\delta, (i+1)\delta]))$, respectively.

The resulting over-approximation is a disjunction of M interval linear relations. In effect, the transcendental relation R_τ in Eq. (1) is over-approximated by an algebraic (bilinear) relation R_τ^I . The over-approximation error be made as small as necessary by increasing the number of subdivisions M , and by using a more expensive procedure for deriving a better approximation of the exponentials by intervals. The problem of evaluating safe interval enclosures to the matrices $e^{[t_1, t_2]A}$ and $P(A, [t_1, t_2])$ uses the idea of scaling and squaring with Horner's rule for evaluating the truncated power series, precisely as described by Goldsztejn [18]. A convenient trick used in our implementation folds the computation of $e^{A, [t_1, t_2]}$ and $P(A, [t_1, t_2])(Bu + b)$ into a single matrix exponential computation for a block matrix of the form $\begin{pmatrix} A & B & b \\ 0 & 0 & 0 \end{pmatrix}$.

Example 3. Consider the hybrid automaton described in Ex. 1. We wish to consider the relational abstraction when τ_1 is taken sometime during the sampling period of 0.2 seconds. To this end, we will choose $M = 2$ and consider two possible intervals for the switching time t when the transition τ_1 is taken $J_1 : [0, 0.1]$ and $J_2 : [0.1, 0.2]$. Considering interval J_1 , we obtain the following relation (intervals are rounded to 2 significant digits for presentation):

$$R_{\tau, J_1} : (\exists \mathbf{y}) \left[\begin{array}{l} \mathbf{x} \in \left(\begin{array}{cc} [0.99, 1.17] & [-0.13, 0.01] \\ [-0.14, 0.0] & [0.98, 1.01] \end{array} \right) \mathbf{y} + \left(\begin{array}{c} [-0.11, 0] \\ [-0.01, 0.05] \end{array} \right) \wedge \\ \mathbf{x}' \in \left(\begin{array}{cc} [1.23, 1.53] & [0.09, 0.25] \\ [0.0, 0.02] & [0.48, 0.7] \end{array} \right) \mathbf{y} + \left(\begin{array}{c} [-0.16, -0.07] \\ [-0.1, -0.04] \end{array} \right) \wedge \mathbf{y} \in G_{\tau_1} \end{array} \right].$$

Templatization. The next step is to use a *templatization* technique to effectively eliminate the quantifiers \mathbf{y}, \mathbf{y}' from the relation R_τ^I in Equation (2) while, at the same time, over-approximating the result by means of a linear arithmetic over-approximation.

Recall that each disjunct in Equation 2 is an *interval linear assertion* of the form

$$R_j^I : (\exists \mathbf{y}, \mathbf{y}') \left(\begin{array}{l} \mathbf{x} \in [\underline{E}_{j,1}, \overline{E}_{j,1}] \mathbf{y} - [\underline{P}_{j,1}, \overline{P}_{j,1}] (B_1 \mathbf{u} + \mathbf{b}_1) \wedge \\ \mathbf{x}' \in [\underline{E}_{j,2}, \overline{E}_{j,2}] \mathbf{y}' - [\underline{P}_{j,2}, \overline{P}_{j,2}] (B_2 \mathbf{u} + \mathbf{b}_2) \wedge \rho_\tau(\mathbf{y}, \mathbf{y}') \end{array} \right),$$

An interval linear constraint of the form $\sum_{j=1}^n I_j x_j + I_0 \leq 0$ is a place holder for a *bi-linear* constraint $\sum_{j=1}^n w_j x_j + w_0 \leq 0$, wherein, w_0, \dots, w_n are freshly introduced variables and each w_j is constrained by requiring that $w_j \in I_j$.

In order to eliminate \mathbf{y}, \mathbf{y}' from this relation, a technique for eliminating quantifiers for real arithmetic such as Cylindrical Algebraic Decomposition (CAD) [9], or a more efficient version for quadratic polynomials is called for [41][13][38]. However, the downside of using such complex techniques include (a) it is well known that QE over non-linear constraints is a hard problem with limited scalability, and (b) the result after elimination will, in general, be a set of polynomial inequalities (semi-algebraic constraint). Therefore, the resulting relationalization may not be easy to reason with for existing tools.

We present a more efficient alternative that side steps the elimination altogether, relying instead on the use of templates and optimization:

1. We choose a set of template expressions $e_k(\mathbf{x}, \mathbf{x}', \mathbf{u})$ involving the variables \mathbf{x} , \mathbf{u} and \mathbf{x}' . We discuss a natural choice for these templates subsequently.
2. For each e_j , we carry out the optimization: $\min e_k$ s.t. $R_j^I(\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{y}', \mathbf{x}')$. If the problem is feasible and bounded, the a_k allows us to conclude that

$$R_j^I(\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{y}', \mathbf{x}') \Rightarrow e_k(\mathbf{x}, \mathbf{u}, \mathbf{x}') \geq a_k.$$

As a result by choosing some $K > 0$ templates e_1, \dots, e_K , we obtain an assertion

$$e_1(\mathbf{x}, \mathbf{u}, \mathbf{x}') \geq a_1 \wedge e_2 \geq a_2 \wedge \dots \wedge e_K(\mathbf{x}, \mathbf{u}, \mathbf{x}') \geq a_K.$$

This assertion serves as an over-approximation to R_j^I with the quantified variables \mathbf{y}, \mathbf{y}' eliminated through optimization. We now provide a natural scheme for the choice of templates, and then discuss how the optimization problem for each template expression can be solved.

The overall relationalization of the plant R_P is the disjunction of the relations R_q , for each mode, and R_τ^I , for each autonomous transition τ .

Theorem 1. *For any pairs of states $\sigma : (q, \mathbf{x})$ and $\sigma' : (q', \mathbf{x}')$ such that \mathbf{x}' is reachable from \mathbf{x} in T_s seconds for constant control input \mathbf{u} , the computed relational abstraction R_P satisfies $R_P(\sigma, \sigma', \mathbf{u})$.*

Choosing Template Expressions. A natural choice for template expression presents itself in our setup by considering the midpoints of the intervals used in the matrix exponential computations. We note that \mathbf{y} is the state obtained starting from \mathbf{x} and evolving in mode 1 for time $[i\delta, (i+1)\delta]$. Likewise, \mathbf{x}' is obtained by evolving according to the state \mathbf{y}' for time $[T_s - (i+1)\delta, T_s - i\delta]$. Finally, $\mathbf{y}' = U(\mathbf{y})$, wherein U is the affine update map for transition τ . In practice, δ is chosen to be small enough to

yield tight enclosures to e^{tA_i} and $P(A_i, t)$ matrices. Therefore, a natural choice of template expression is obtained by considering the midpoints of the time intervals involved. Specifically, we consider the affine expressions defined by

$$\mathbf{x}' = e^{(T_s - t_m)A_2} U(e^{t_m A_1} \mathbf{x}), \text{ where } t_m = (i + \frac{1}{2})\delta.$$

Example 4. In Ex. 3, we showed the interval linear relation obtained by considering switching times in the interval $t \in [0.0, 0.1]$. The midpoint of this interval is $t_m = 0.05$. Therefore, we consider the mode n_0 taken for time 0.05 units followed by 0.15 units of mode 1 for generating a suitable template. These template expressions are given by $e_1 : x' - 1.31x - 0.25y$ and $e_2 : y' - 0.05x - 0.6y$. We seek to bound these expressions to obtain a linear arithmetic over-approximation.

Encoding Optimization. Next, we turn our attention to setting up the optimization problem for a given template expression $\mathbf{c}\mathbf{x} + \mathbf{d}\mathbf{x}'$. The intermediate states \mathbf{y}, \mathbf{y}' are related by interval linear expressions of the form

$$\mathbf{x}' \in [\underline{E}_2, \overline{E}_2]\mathbf{y}' + [\underline{P}_2, \overline{P}_2], \quad \mathbf{x} \in [\underline{E}_1, \overline{E}_1]\mathbf{y} + [\underline{P}_1, \overline{P}_1].$$

To set up the optimization problem, we substitute these expressions for \mathbf{x}, \mathbf{x}' in the template to obtain $\mathbf{c}([\underline{E}_1, \overline{E}_1]\mathbf{y} + [\underline{P}_1, \overline{P}_1]) + \mathbf{d}([\underline{E}_2, \overline{E}_2]\mathbf{y}' + [\underline{P}_2, \overline{P}_2])$. This is, in fact, an interval linear expression involving \mathbf{y}, \mathbf{y}' . The overall optimization problem reduces to: $\min [\underline{\mathbf{c}}, \overline{\mathbf{c}}]\mathbf{y} + [\underline{\mathbf{d}}, \overline{\mathbf{d}}]\mathbf{y}' + [\underline{c}_0, \overline{c}_0]$ s.t. $\rho_\tau(\mathbf{y}, \mathbf{y}')$. Here $[\underline{\mathbf{c}}, \overline{\mathbf{c}}] = \mathbf{c}[\underline{E}_1, \overline{E}_1]$, $[\underline{\mathbf{d}}, \overline{\mathbf{d}}] = \mathbf{d}[\underline{E}_2, \overline{E}_2]$ and $[\underline{c}_0, \overline{c}_0] = \mathbf{c}[\underline{P}_1, \overline{P}_1] + \mathbf{d}[\underline{P}_2, \overline{P}_2]$. The problem has an interval linear objective and linear constraints. We now show that the constraints can be encoded into a disjunctive linear program.

Theorem 2. *The optimization of an interval linear objective w.r.t linear constraints*

$$\min [\underline{\mathbf{c}}, \overline{\mathbf{c}}] \times \mathbf{z} \text{ s.t. } A\mathbf{z} \leq \mathbf{b},$$

can be equivalently expressed as a linear program with disjunctive constraints:

$$\min \underline{\mathbf{c}}\mathbf{z}^+ - \overline{\mathbf{c}}\mathbf{z}^- \text{ s.t. } A\mathbf{z}^+ - A\mathbf{z}^- \leq \mathbf{b}, \quad \mathbf{z}^+, \mathbf{z}^- \geq 0, \quad z_i^+ = 0 \vee z_i^- = 0$$

where $\mathbf{z} = \mathbf{z}^+ - \mathbf{z}^-$.

Proof. We may decompose any vector \mathbf{z} as $\mathbf{z} = \mathbf{z}^+ - \mathbf{z}^-$, where $\mathbf{z}^+, \mathbf{z}^- \geq 0$, and enforce $z_i^+ z_i^- = 0$. Next, consider the objective $[\underline{\mathbf{c}}, \overline{\mathbf{c}}] \times (\mathbf{z}^+ - \mathbf{z}^-)$. Since correspondent entries in $\mathbf{z}^+, \mathbf{z}^-$ cannot be positive at the same time, we may write the objective as a linear expression $\underline{\mathbf{c}}\mathbf{z}^+ - \overline{\mathbf{c}}\mathbf{z}^-$. Finally, the complementarity condition $z_i^+ z_i^- = 0$ is rewritten as $z_i^+ = 0 \vee z_i^- = 0$.

A simple approach to solve the optimization problem for disjunctive constraints is to use a linear arithmetic SMT solver to repeatedly obtain feasible solutions $\mathbf{z}^+, \mathbf{z}^-$. For a given feasible solution output by the SMT solver, we fix a minimal set of the values for $\mathbf{z}^+, \mathbf{z}^-$ to zero to enforce the complementarity constraints $z_i^+ z_i^- = 0$, leaving the

remaining variables as unknowns. An LP solver is then used to compute an optimal value f^* for the objective function f , based on the remaining constraints. This yields a potential optimum. Next, we add a *blocking constraint* $f > f^*$ to the SMT solver and search for a different solution. The process is carried out until the SMT solver returns UNSAT. At this point, we output the last optimal solution as the final value.

Example 5. Continuing with the examples worked out in Ex. [3] we perform the optimization of the template expressions chosen in Ex. [4] to obtain the relational abstraction:

$$R_{\tau_1, J_1} : -1.0 \leq 1.31x + 0.25y - x' \leq 1.24 \wedge -0.32 \leq 0.05x + 0.6y - y' \leq 0.51.$$

Likewise, considering the time interval $J_2 : [0.1, 0.2]$ for the switching time, we obtain the abstraction:

$$R_{\tau_1, J_2} : -1 \leq 0.94x + 0.25y - x' \leq 0.88 \wedge -0.54 \leq 0.17x + 0.9y - y' \leq 0.7708.$$

The overall timed relational abstraction for the sampling period where τ_1 can be taken sometime in between is $R_{\tau_1} : R_{\tau_1, J_1} \vee R_{\tau_1, J_2}$.

4 Experimental Evaluation

We first briefly describe our implementation of the relational abstractor using the techniques presented here.

Implementation: The relational abstractor takes in a plant description including the sample time T_s , and outputs the relation as a SAL transition system [32]. The relationalization is performed for the continuous dynamics in each mode by computing a matrix exponential. A numerical approximation of the matrix exponential function is obtained using Pade's approximation [27]. We have also implemented a procedure that provides a sound interval enclosure of the exponential function over interval matrices using the ideas described by Goldsztejn [18]. However, this procedure is used solely for dealing with autonomous transitions.

Autonomous transition between modes are handled using the algorithms presented so far. We implicitly assume minimum dwell time greater than or equal to the sampling time for the controller. The optimization problems encountered for autonomous transitions are solved using the SMT solver Z3 [12]. SAL provides a k -induction and BMC engine using the solver Yices [14]. This was used for analyzing the resulting composed transition system for our evaluations.

Benchmarks: Table 1 shows the benchmarks used in our evaluation along with their sources. The benchmarks vary in dimensionality (column #Var) and number of transitions (column #Trs). Note that many benchmarks do not contain autonomous transitions. For each benchmark, we performed the relational abstraction for different sampling times T_s , and used SAL to analyze safety properties (column Prop.).

The NAV benchmarks, due to Ivancic and Fehnker [15], model a particle traveling through many 2D cells that each have a different dynamics. We consider two versions of this benchmark (a) the transitions in the benchmark are all interpreted as controlled, commanded by a controller, or (b) transitions are autonomous in nature. Starting with

Table 1. The benchmarks used in our experiments at a glance

Model	Description	# Var	# Mode	# Trs	Prop.	Description
InvPen	Inverted Pendulum Control	5	1	1	$\theta_b(0.05)$	Angle $\theta \in [-0.05, 0.05]$
SNCS	Network Control System [42]	2	1	2	P_1 P_2	$(x, y) \in [-100, 100]^2$ $(x, y) \in [-10^4, 10^4]^2$
ACC	Adaptive Cruise Control [21]	4	1	2	SAFE	No collision between cars.
ACC-T	ACC + transmission [21]	3	20	24	SAFE	No collision between cars.
Heat-x	Room heater [15]	9	8	20	LB	Lower bounds on temp. Cf. [15]
Nav-y	NAV benchmarks [15]	4	[7,16]	[9,16]	RA RB	Cell A is unreachable Cell B is unreachable
Toy	Example [1]	2	2	5	$bnd(k)$	$n_1 \Rightarrow \mathbf{x} \leq k$
Ring(n,m)	Cf. Section 4	n	m+1	m+1	$bnd(k)$	$n_4 \Rightarrow \mathbf{x} \leq k$

all controlled transitions, we introduce uncontrolled transitions incrementally into these benchmarks.

Ring Benchmarks: We created a set of sampled data control systems with autonomous transitions. We consider a plant with $k + 1$ modes, wherein modes m_1, \dots, m_k are governed by stable dynamics, while mode m_{k+1} is an unstable mode. The controller seeks to stabilize this mode by periodically sensing the plant’s state and applies a control that reverts it back to mode m_1 .

The benchmark instance Ring (n, k) consists of n state variables and $k + 1$ modes. The autonomous transitions are added from mode i to mode $i + 1$ for $i \leq k$, while the controlled transition leads from mode $k + 1$ to mode 1. The dynamics in each mode is of the form $\frac{d\mathbf{x}}{dt} = A_i(\mathbf{x} - \mathbf{b}_i)$, wherein, for the stable modes A_i is a Hurwitz matrix and \mathbf{b}_i is a designated equilibrium for m_i . For the unstable mode, we ensure that A_i has a positive eigenvalue. The switch from m_i to m_{i+1} takes place inside a box $[b_i - \epsilon, b_i + \epsilon]$. The controller periodically senses the plant and whenever $|\mathbf{x}| > c$ for some fixed c , it brings the dynamics back into the box $|\mathbf{x}| < c$ while transitioning to mode m_1 . We wish to check whether all trajectories lie inside a box $|\mathbf{x}| \leq c + d$, for varying tolerances d .

Results: Table 2 shows the experimental results on benchmarks that do not have autonomous transitions. Our experiments are attempted using numerous values of sample times (T_s) for each property until either a proof is obtained for the controller or the SAL tool fails due to a timeout. In the absence of autonomous transitions, the relationalization time for all these benchmarks was well under 1 second. We also note that the counterexamples generated by SAL can be concretized, since the timed abstractions involving matrix exponentials are seen to be quite precise.

Table 3 shows the results for systems with controlled and autonomous transitions. These include the system from Ex. 1, the Ring (n, k) systems for varying n and the NAV benchmark instances as we increase the number of autonomous transitions. We observe that making all the transitions autonomous leads to a counterexample. This counterexample may potentially be an artifact of the precision loss due to relationalization of autonomous transitions. Future work will consider the refinement of these counterexamples by subdividing the transition switching time intervals further based on spurious counterexamples. We note that the time for relationalization remains a small

Table 2. Results on benchmarks without autonomous transitions. All timings were measured in seconds on a laptop running Intel Core i7-2820Q 2.30GHz processor (x86_64 arch) with 8GB RAM running Ubuntu 11.04 Linux 2.6.38-13. Legend: **CE** indicates true counter-example, **P** indicates proofs, **F** indicates failure due to timeout.

Model	Prop	T_s	Result	Depth	Time
InvPen	$\theta_b(0.05)$	0.1	CE	1	0.1
		0.05	P	12	0.9
SNCS	P_1	1.7	P	2	0.1
	P_2	1.8	CE	93	5.6
ACC	SAFE	0.1	P	7	0.1
	gap(100)	0.1	P	4	0.1
ACC-T	SAFE	1	P	14	2.2
Nav 1-7	RB	1	P	<11	<5
	RA	1	CE	<13	<2
Nav 8	RB	1	CE	7	0.27
	RB	0.2	F	25	> 1h
Nav 9	RB	1	P	19	213.05
	RA	1	CE	9	0.37
Nav 10	RB	1	CE	19	28.37
	RB	0.5	F	25	> 1h

Model	Prop	T_s	result	depth	time
Heat1	LB	1	CE	4	0.1
		.2	CE	8	0.1
		0.1	P	37	1967
Heat2	LB	1	CE	4	0.1
		0.2	P	17	160
		0.1	CE	2	0.1
Heat3	LB	1	CE	2	0.1
		0.2	CE	17	27
		0.1	F	30	> 1h
Heat4	LB	1	CE	2	0.1
		0.1	CE	10	1.22
		0.02	F	25	> 1h

fraction of the time needed to check the system. The relationalization scheme can be improved further if SMT solvers such as Z3 can be modified to support the optimization of objective functions.

Comparison with SpaceEx: We now compare the results obtained for our approach with the SpaceEx tool over the same set of benchmarks [16]. While performing the comparison with SpaceEx, we reiterate two key points of difference: (a) SpaceEx handles general hybrid systems with support for synchronous time-triggered semantics as well as the standard event-triggered semantics given by guards and resets. Our technique is specialized to sampled data control systems. (b) SpaceEx attempts to characterize the reachable sets for all time instances, whereas our approach focuses on proving properties at the periodic sampling times.

Typically, running the benchmarks in SpaceEx required choosing from a range of parameters including template domains, underlying implementation, flowpipe tolerances, error tolerances, local and global time horizons and limits on the number of iteration. We ran SpaceEx for each benchmark using multiple option sets, choosing the option that provided the “best answer” with as few warnings as possible. However, it may be possible to obtain qualitatively different results using choices for the parameters that were unexplored in our experiments. A detailed table summarizing our experiences is available upon request.

Table 4 presents a summary of the results obtained by running SpaceEx on our benchmarks. We note that in many cases, SpaceEx did not reach a fixed point. Therefore, whenever a property proof was obtained, we report if the proof was obtained over a finite time horizon. Likewise, for cases where the property was not proved, we ran SpaceEx for the minimum number of iterations until a potential violation is observed.

Table 3. Results on systems with autonomous transitions. For the NAV benchmarks, autonomous transitions between cells were incrementally enabled over the controlled transitions until all transitions were autonomous. T_{sal} refers to running time for SAL and T_{rel} the running time for the relationalization.

Model	Prop	result	depth	T_{sal}	T_{rel}		Model	Prop	T_s	# Aut.	result	depth	time		
Toy	$bnd(8)$	P	2	0.1	< .1		Nav1	RB	0.2	6	P	9	199		
	$bnd(6)$	P	2	0.1						14	P	9	72		
	$bnd(5.5)$	P	2	0.3						21	P	9	96		
	$bnd(5)$	CE	70	204						24	F	9	169		
Ring(3,4)	$bnd(20)$	P	10	5.2	0.8		Nav2	RB	0.2	20	P	9	92		
	$bnd(15)$	P	30	451						21	F	9	160		
Ring(5,4)	$bnd(25)$	P	10	34.7	2.8					All	CE	9	161		
	$bnd(20)$	P	10	56.2			Nav3	RB	0.2	22	P	9	83		
	$bnd(15)$	F	20	> 1h						All	CE	6	13		
Ring(7,4)	$bnd(25)$	P	10	157	11.9		Nav4	RB	0.2	9	P	18	1305		
	$bnd(20)$	P	10	357						20	F	18	> 1h		
	$bnd(15)$	F	20	> 1h						All	CE	6	7		
Ring(9,4)	$bnd(25)$	P	10	515	19.1										
	$bnd(20)$	P	10	2929											
Ring(11,4)	$bnd(25)$	F	10	> 1h		150									

Table 4. Results of SpaceEx tool on the benchmark. Legend: **A:** All transitions were Autonomous, **C:** All transitions were controlled Autonomous, **Prop:** property, **F:** Found potentially spurious counter-example (our approach proves model+property), **P:** Proved, **FT:** proof valid over a finite time horizon, **Time:** Running time in seconds.

Model	Prop	T_s	Result	Time	Model	Prop	T_s	Result	Time
InvPen	$\theta_b(0.05)$	0.05	F	6	Nav 4(A)	RB	—	P (FT)	1501
SNCS	P_1	1.7	F	371	Nav 6(A)	RB	—	F	1223
Heat1	LB	0.1	F	557	Nav 7(A)	RB	—	P (FT)	615
Ring(3,4)	$bnd(20)$	0.2	P (FT)	3475	Nav 9(A)	RB	—	F	619
Ring(5,4)	$bnd(25)$	0.2	P (FT)	2051	Nav 4(C)	RB	1	P (FT)	109
Ring(7,4)	$bnd(25)$	0.2	F	6323	Nav 6(C)	RB	1	P (FT)	167
Ring(9,4)	$bnd(25)$	0.2	F	709	Nav 7(C)	RB	1	P	7
					Nav 9(C)	RB	1	F	6

The comparison between our approaches clearly showcases some of the relative merits and demerits of our approach vis-a-vis SpaceEx. There are many benchmarks wherein our approach is able to establish the property over an infinite time horizon using k -induction, whereas SpaceEx either proves the property over a finite time horizon or fails. On the other hand, the NAV benchmarks are an interesting case where SpaceEx’s performance is at par or clearly superior to that of our approach.

For some of the Ring examples, we observed that using the bounds obtained by SpaceEx as inductive strengthenings enabled the k -induction technique to prove the property for a smaller value of k , leading to improved running times.

Our future work will focus on an integration of the approaches considered here in combination with tools such as SpaceEx to achieve infinite horizon safety property proofs. Another important area of future research will be to extend our approach to analyze non linear hybrid systems, which are much more challenging.

References

1. Alur, R., Dang, T., Ivančić, F.: Counter-Example Guided Predicate Abstraction of Hybrid Systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 208–223. Springer, Heidelberg (2003)
2. Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O.: Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. In: Proc. High Confidence Medical Devices, Software Systems and Medical Device Plug and Play Interoperability (2007)
3. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. *Acta Informatica* 43, 451–476 (2007)
4. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O’Hearn, P.W.: Variance analyses from invariance analyses. In: POPL, pp. 211–224. ACM (2007)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
6. Bochev, P., Markov, S.: A self-validating numerical method for the matrix exponential. *Computing* 43(1), 59–72
7. Bozzano, M., Cimatti, A., Tapparo, F.: Symbolic Fault Tree Analysis for Reactive Systems. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 162–176. Springer, Heidelberg (2007)
8. 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)
9. Collins, G.: Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
10. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Principles of Programming Languages, pp. 238–252 (1977)
11. Dang, T., Maler, O., Testylier, R.: Accurate hybridization of nonlinear systems. In: HSCC 2010, pp. 11–20. ACM (2010)
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)
13. Dolzmann, A., Sturm, T.: REDLOG: Computer algebra meets computer logic. *ACM SIGSAM Bulletin* 31(2), 2–9 (1997)
14. Dutertre, B., de Moura, L.: The YICES SMT solver. Cf, <http://yices.csl.sri.com/tool-paper.pdf> (last viewed January 2009)
15. Fehnker, A., Ivančić, F.: Benchmarks for Hybrid Systems Verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004)
16. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)

17. Girard, A.: Reachability of Uncertain Linear Systems Using Zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005)
18. Goldsztejn, A.: On the exponentiation of interval matrices. Preprint (Working Paper) # hal-00411330, version 1. Cf (2009),
<http://hal.archives-ouvertes.fr/hal-00411330/fr/>
19. Guernic, C.L., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Analysis: Hybrid Systems 4(2), 250–262 (2010)
20. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI (2009)
21. Gulwani, S., Tiwari, A.: Constraint-Based Approach for Analysis of Hybrid Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
22. Halbwachs, N., Proy, Y.-E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11(2), 157–185 (1997)
23. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. IEEE Transactions on Automatic Control 43, 540–554 (1998)
24. Jeannet, B., Halbwachs, N., Raymond, P.: Dynamic Partitioning in Analyses of Numerical Properties. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 39–50. Springer, Heidelberg (1999)
25. Kurzhanski, A.B., Varaiya, P.: Ellipsoidal Techniques for Reachability Analysis. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 202–214. Springer, Heidelberg (2000)
26. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
27. Moler, C., Loan, C.V.: Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. SIAM Review 45(1), 161–208 (2003)
28. Moore, R., Kearfott, R.B., Cloud, M.: Introduction to Interval Analysis. SIAM (2009)
29. Oppenheimer, E.P., Michel, A.N.: Application of interval analysis techniques to linear systems. II. the interval matrix exponential function. IEEE Trans. on Circuits and Systems 35(10), 1230–1242 (1988)
30. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society (2004)
31. Podelski, A., Wagner, S.: Model Checking of Hybrid Systems: From Reachability Towards Stability. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 507–521. Springer, Heidelberg (2006)
32. Rushby, J., Lincoln, P., Owre, S., Shankar, N., Tiwari, A.: Symbolic analysis laboratory (sal). Cf, <http://www.cs1.sri.com/projects/sal/>
33. Sankaranarayanan, S., Dang, T., Ivančić, F.: Symbolic Model Checking of Hybrid Systems Using Template Polyhedra. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 188–202. Springer, Heidelberg (2008)
34. Sankaranarayanan, S., Homaei, H., Lewis, C.: Model-Based Dependability Analysis of Programmable Drug Infusion Pumps. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 317–334. Springer, Heidelberg (2011)
35. Sankaranarayanan, S., Tiwari, A.: Relational Abstractions for Continuous and Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 686–702. Springer, Heidelberg (2011)
36. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)

37. Silva, B.I., Richeson, K., Krogh, B.H., Chutinan, A.: Modeling and verification of hybrid dynamical system using checkmate. In: ADPM 2000 (2000),
<http://www.ece.cmu.edu/~webk/checkmate>
38. Sturm, T., Tiwari, A.: Verification and synthesis using real quantifier elimination. In: ISSAC, pp. 329–336. ACM (2011)
39. Tiwari, A.: Abstractions for hybrid systems. Formal Methods in Systems Design 32, 57–83 (2008)
40. Tiwari, A., SRI International: HybridSAL: A tool for abstracting HybridSAL specifications to SAL specifications. Cf (2007) <http://sal.csl.sri.com/hybridsal/>
41. Weispfenning, V.: Quantifier elimination for real algebra—the quadratic case and beyond. In: Applied Algebra and Error-Correcting Codes (AAECC), vol. 8, pp. 85–101 (1997)
42. Zhang, W., Branicky, M.S., Phillips, S.M.: Stability of networked control systems. IEEE Control Systems Magazine 21, 84–99 (2001)

Approximately Bisimilar Symbolic Models for Digital Control Systems

Rupak Majumdar¹ and Majid Zamani²

¹ Max Planck Institute for Software Systems, Germany
rupak@mpi-sws.org

² Department of Electrical Engineering, University of California at Los Angeles
zamani@ee.ucla.edu

Abstract. Symbolic approaches to control hybrid systems construct a discrete approximately-bisimilar abstraction of a continuous control system and apply automata-theoretic techniques to construct controllers enforcing given specifications. For the class of digital control systems (i.e., whose control signals are piecewise constant) satisfying incremental input-to-state stability (δ -ISS), existing techniques to compute discrete abstractions begin with a quantization of the state and input sets, and show that the quantized system is approximately bisimilar to the original if the sampling time is sufficiently large or if the Lyapunov functions of the system decrease fast enough. If the sampling time is not sufficiently large, the former technique fails to apply. While abstraction based on Lyapunov functions may be applicable, because of the conservative nature of Lyapunov functions in practice, the size of the discrete abstraction may be too large for subsequent analyses.

In this paper, we propose a technique to compute discrete approximately-bisimilar abstractions of δ -ISS digital control systems. Our technique quantizes the state and input sets, but is based on multiple sampling steps: instead of requiring that the sampling time is sufficiently large (which may not hold), the abstract transition system relates states multiple sampling steps apart.

We show on practical examples that the discrete state sets computed by our procedure can be several orders of magnitude smaller than existing approaches, and can compute symbolic approximate-bisimilar models even when other existing approaches do not apply or time-out. Since the size of the discrete state set is the main limiting factor in the application of symbolic control, our results enable symbolic control of larger systems than was possible before.

1 Introduction

Many cyber-physical systems involve the complex interplay between continuous controlled dynamical systems and discrete controllers. Correctness requirements for these systems involve temporal specifications about the evolution of the dynamics, which are not easily amenable to classical continuous controller synthesis techniques. As a result, in recent years, a lot of research has focused on *symbolic*

models of systems involving both continuous and discrete components (so called *hybrid systems*). A symbolic model is a discrete approximation of the continuous system such that controllers designed for the approximation can be refined to controllers for the original system. Symbolic models are interesting because they allow the algorithmic machinery for controller synthesis of discrete systems w.r.t. temporal specifications [5][22][14] to be used to automatically synthesize controllers for hybrid systems.

The key to this methodology is the existence of finite-state symbolic models that are *bisimilar* to the original system. When the continuous time model has very simple dynamics, such as clocks, one can show that finite-state bisimilar models exist and can be effectively computed [2]. Unfortunately, even for very simple dynamics such as stopwatches, finite-state bisimilar models may not exist [12]. The insight, developed over the past five years, is that for control of dynamical systems where there is a natural metric on the states, one can use ε -*approximations* of classical equivalence relations [8][9] to relate the original system and the symbolic model. An ε -approximate bisimulation relaxes the condition of bisimulation by requiring that two bisimilar states are within ε distance of each other, and guarantees that for each trajectory starting at one state, there is a trajectory starting at the other that is always within ε distance away. For many continuous dynamical systems, ε -approximate bisimulation relations of finite index have been shown to exist [20][10][17][18].

We focus on digital control systems, in which there is a fixed sampling time τ and the control action is chosen from a compact set and held constant for τ time units. Current approaches to building the symbolic model, such as [17][18][10], proceed as follows. First, they choose discretizations of the state and input sets. Then, they use either the incremental stability assumption or incremental Lyapunov functions to show that if the discretizations are sufficiently small, and the sampling time τ is sufficiently big, then the resulting discrete abstraction is ε -approximate bisimilar to the original system. If the sample time, which is usually not under the control of the verification engineer, is not sufficiently large, the technique will not apply. Even if the method applies, the resulting state space is often prohibitively large. This is usually the case for symbolic models built using conservative Lyapunov functions [10].

We show a construction of approximately bisimilar models for digital control systems that improves upon known algorithms. The insight in our construction is to consider a number of sampling steps instead of only one step. That is, we dilate the quantum of time of the control system and observe the system only every k steps, for some parameter k . Then, instead of requiring that the sampling time is sufficiently big, we only require that the number of steps is chosen sufficiently large, so that the technique is always applicable. Further, we demonstrate experimentally that our technique can give symbolic models that require a much coarser discretization of the state and input sets, resulting in symbolic models with many fewer states, while guaranteeing ε -approximate bisimulation with the original system.

We have implemented our algorithm on top of Pessoa [16], a tool that computes symbolic models and then performs controller synthesis on the symbolic models. We show on a set of benchmark examples that our technique produces symbolic models whose state sets are orders of magnitude smaller than previous approaches, and can finish computing the symbolic model and performing controller synthesis in seconds, when previous techniques either do not apply, or time-out after several hours.

2 Systems and Approximate Equivalences

Preliminaries: The symbols \mathbb{N} , \mathbb{Z} , \mathbb{R} , \mathbb{R}^+ , and \mathbb{R}_0^+ denote the set of natural, integer, real, positive, and non-negative real numbers, respectively. A *metric space* (Y, d) consists of a set Y and a metric $d : Y \times Y \rightarrow \mathbb{R}_0^+$ on Y . Given a vector $x \in \mathbb{R}^n$, we denote by x_i the i th element of x , and by $\|x\|$ the infinity norm of x , namely, $\|x\| = \max\{|x_1|, |x_2|, \dots, |x_n|\}$. Given a matrix $M = \{m_{ij}\} \in \mathbb{R}^{n \times m}$, we denote by $\|M\|$ the infinity norm of M , namely, $\|M\| = \max_{1 \leq i \leq n} \sum_{j=1}^m |m_{ij}|$. The symbol I_n denotes the identity matrix in $\mathbb{R}^{n \times n}$. The closed ball centered at $x \in \mathbb{R}^n$ with radius ε is defined by $\mathcal{B}_\varepsilon(x) = \{y \in \mathbb{R}^n \mid \|x - y\| \leq \varepsilon\}$.

Given a measurable function $f : \mathbb{R}_0^+ \rightarrow \mathbb{R}^n$, the (*essential*) supremum (sup norm) of f is denoted by $\|f\|_\infty$; we recall that $\|f\|_\infty = (\text{ess sup}) \sup \{\|f(t)\|, t \geq 0\}$. A continuous function $\gamma : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ is said to belong to class \mathcal{K} if it is strictly increasing and $\gamma(0) = 0$; γ is said to belong to class \mathcal{K}_∞ if $\gamma \in \mathcal{K}$ and $\gamma(r) \rightarrow \infty$ as $r \rightarrow \infty$. A continuous function $\beta : \mathbb{R}_0^+ \times \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ is said to belong to class \mathcal{KL} if, for each fixed s , the map $\beta(r, s)$ belongs to class \mathcal{K}_∞ with respect to r and, for each fixed nonzero r , the map $\beta(r, s)$ is decreasing with respect to s and $\beta(r, s) \rightarrow 0$ as $s \rightarrow \infty$.

A set $B \subseteq \mathbb{R}^n$ is called a *box* if $B = \prod_{i=1}^n [c_i, d_i]$, where $c_i, d_i \in \mathbb{R}$ with $c_i < d_i$ for each $i \in \{1, \dots, n\}$. The *span* of a box B is defined as $\text{span}(B) = \min\{|d_i - c_i| \mid i = 1, \dots, n\}$. For a box B and $\eta \leq \text{span}(B)$, define the η -approximation $[B]_\eta = \{b \in B \mid b_i = k_i \eta \text{ for some } k_i \in \mathbb{Z}, i = 1, \dots, n\}$. Note that $[B]_\eta \neq \emptyset$ for any $\eta \leq \text{span}(B)$. Geometrically, for any $\eta \in \mathbb{R}^+$ with $\eta \leq \text{span}(B)$ and $\lambda \geq \eta$ the collection of sets $\{\mathcal{B}_\lambda(p)\}_{p \in [B]_\eta}$ is a finite covering of B , i.e., $B \subseteq \bigcup_{p \in [B]_\eta} \mathcal{B}_\lambda(p)$. By defining $[\mathbb{R}^n]_\eta = \{a \in \mathbb{R}^n \mid a_i = k_i \eta, k_i \in \mathbb{Z}, i = 1, \dots, n\}$, the set $\bigcup_{p \in [\mathbb{R}^n]_\eta} \mathcal{B}_\lambda(p)$ is a countable covering of \mathbb{R}^n for any $\eta \in \mathbb{R}^+$ and $\lambda \geq \eta/2$. We extend the notions of span and approximation to finite unions of boxes as follows. Let $A = \bigcup_{j=1}^M A_j$, where each A_j is a box. Define $\text{span}(A) = \min\{\text{span}(A_j) \mid j = 1, \dots, M\}$, and for any $\eta \leq \text{span}(A)$, define $[A]_\eta = \bigcup_{j=1}^M [A_j]_\eta$.

Digital Control Systems: We now define digital control systems, which are continuous time controlled dynamical systems with piecewise-constant inputs of a fixed duration.

Definition 1. A (digital) control system is a tuple $\Sigma = (\mathbb{R}^n, \tau, \mathbf{U}, \mathcal{U}_\tau, f)$ where:

- \mathbb{R}^n is the state space;

- $\tau \in \mathbb{R}^+$ is the sampling time;
- $\mathbf{U} \subseteq \mathbb{R}^m$ is the input set, which is assumed to be a finite union of boxes containing the origin;
- \mathcal{U}_τ contains curves piecewise constant of duration τ , i.e.,

$$\mathcal{U}_\tau = \left\{ v : \mathbb{R}_0^+ \rightarrow \mathbf{U} \mid v(t) = v((l-1)\tau), t \in [(l-1)\tau, l\tau[, l \in \mathbb{N} \right\};$$

- $f : \mathbb{R}^n \times \mathbf{U} \rightarrow \mathbb{R}^n$ is a continuous map satisfying the following Lipschitz assumption: for every compact set $Q \subset \mathbb{R}^n$, there exists a constant $L \in \mathbb{R}^+$ such that for all $x, y \in Q$ and all $u \in \mathbf{U}$, we have: $\|f(x, u) - f(y, u)\| \leq L\|x - y\|$.

A curve $\xi :]a, b[\rightarrow \mathbb{R}^n$ is said to be a *trajectory* of Σ if there exists $v \in \mathcal{U}_\tau$ satisfying:

$$\dot{\xi}(t) = f(\xi(t), v(t)), \quad (1)$$

for almost all $t \in]a, b[$. We also write $\xi_{xv}(t)$ to denote the point reached at time t under the input v from initial condition $x = \xi_{xv}(0)$; this point is uniquely determined, since the assumptions on f ensure existence and uniqueness of trajectories [19]. Although we have defined trajectories over open domains, we shall refer to trajectories $\xi_{xv} : [0, \tau] \rightarrow \mathbb{R}^n$ and input curves $v : [0, \tau[\rightarrow \mathbf{U}$, with the understanding of the existence of a trajectory $\xi'_{xv'} :]a, b[\rightarrow \mathbb{R}^n$ and input curve $v' :]a, b[\rightarrow \mathbf{U}$ such that $\xi_{xv} = \xi'_{xv'}|_{[0, \tau]}$ and $v = v'|_{[0, \tau]}$. Note that by continuity of ξ , we have that $\xi_{xv}(\tau)$ is uniquely defined as the left limit of $\xi_{xv}(t)$ with $t \rightarrow \tau$.

A control system Σ is said to be *forward complete* if every trajectory is defined on an interval of the form $]a, \infty[$. Sufficient and necessary conditions for a system to be forward complete can be found in [4].

Stability: As in [17][18][10], we assume incremental input-to-state stability of the control system.

Definition 2. [3] A control system Σ is incrementally input-to-state stable (δ -ISS) if it is forward complete and there exist a \mathcal{KL} function β and a \mathcal{K}_∞ function γ such that for any $t \in \mathbb{R}_0^+$, any $x, x' \in \mathbb{R}^n$, and any $v, v' \in \mathcal{U}_\tau$ the following condition is satisfied:

$$\|\xi_{xv}(t) - \xi_{x'v'}(t)\| \leq \beta(\|x - x'\|, t) + \gamma(\|v - v'\|_\infty). \quad (2)$$

Remark 1. For linear control systems, the functions β and γ in Definition 2 can be explicitly computed as follows. It can be readily verified that any linear control system:

$$\dot{\xi} = A\xi + Bv, \quad \xi(t) \in \mathbb{R}^n, \quad v(t) \in \mathbf{U} \subseteq \mathbb{R}^m, \quad (3)$$

is δ -ISS if and only if A is globally asymptotically stable, i.e., every eigenvalue of A has strictly negative real part. Then, the functions β and γ can be chosen as:

$$\beta(r, t) = \|e^{At}\| r; \quad \gamma(r) = \left(\|B\| \int_0^\infty \|e^{As}\| ds \right) r, \quad (4)$$

where $\|e^{At}\|$ denotes the infinity norm of e^{At} .

The assumption of δ -ISS does not restrict the class of digital control systems significantly. If a control system Σ is not δ -ISS, one can design an internal control loop rendering Σ δ -ISS. Assume there exists a smooth controller $k(\xi, v)$ rendering control system Σ δ -ISS with respect to the input v . Note that by using the results in [21], there exists a positive number τ^* such that for any $\tau < \tau^*$, the digitized version of k makes the system Σ δ -ISS with respect to the input v . Now, one can drive v by designing another controller on top of k to satisfy some desired specifications, e.g., using the symbolic abstraction techniques explained in Section 3.

If Σ is a linear control system, one can use the results in control theory [13] to design a state feedback gain k rendering Σ globally asymptotically stable and, hence, δ -ISS. If Σ is a nonlinear control system, one can use the results in [23] to design controllers rendering Σ δ -ISS.

Metric Systems and Approximate Relations: We now recall the notions of metric systems that we will use as abstract models for control systems as well as approximate bisimulation relations on metric systems that will be central to our abstractions.

Definition 3. A metric system S is a quintuple $S = (X, U, \rightarrow, (Y, d), H)$ consisting of a (possibly infinite) set of states X ; a set of inputs U ; a transition relation $\rightarrow \subseteq X \times U \times X$; an output metric space (Y, d) ; and an output function $H : X \rightarrow Y$.

A transition $(x, u, x') \in \rightarrow$ is written by $x \xrightarrow{u} x'$. If $x \xrightarrow{u} x'$, state x' is called a u -successor, or simply, successor, of state x . A metric system is *countable* (resp. *finite*) if X is countable (resp. finite). A metric system is *deterministic* if for any state $x \in X$ and any input $u \in U$, there exists at most one u -successor (there may be none).

Metric systems capture the dynamics of a system through the transition relation: for states $x, x' \in X$ and $u \in U$, if $x \xrightarrow{u} x'$ then it is possible to evolve from state x to state x' under the input labeled by u . Given a control system $\Sigma = (\mathbb{R}^n, \tau, U, \mathcal{U}_\tau, f)$, define the metric system $S(\Sigma) = (\mathbb{R}^n, \mathcal{U}_\tau, \rightarrow, (\mathbb{R}^n, \|\cdot\|), \lambda x. x)$ where $x \xrightarrow{u} x'$ if there is a trajectory $\xi_{xu} : [0, \tau] \rightarrow \mathbb{R}^n$ such that $x' = \xi_{xu}(\tau)$, and (with abuse of notation) we define the metric $\|\cdot\| : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}_0^+$ as $\|x - y\|$ for any $x, y \in \mathbb{R}^n$.

Intuitively, the metric system captures all the behaviors of the original control system. The notion of capturing all behaviors is formalized using (approximate) bisimulation relations [15, 9], that we define below.

Definition 4. Let $S = (X, U, \rightarrow, (Y, d), H)$ and $S' = (X', U', \rightarrow', (Y', d'), H')$ be metric systems with the same output metric spaces $(Y, d) = (Y', d')$. For each $\varepsilon \geq 0$, a binary relation $R \subseteq X \times X'$ is said to be an ε -approximate simulation relation from S to S' , if the following three conditions are satisfied:

- (i) for every $x \in X$, there exists $x' \in X'$ with $(x, x') \in R$;
- (ii) for every $(x, x') \in R$ we have $d(H(x), H'(x')) \leq \varepsilon$;

(iii) for every $(x, x') \in R$ and $u \in U$, if $x \xrightarrow{u} y$ in S then there exists some $u' \in U'$ such that $x' \xrightarrow{u'} y'$ and $(y, y') \in R$.

System S is ε -approximately simulated by S' , denoted by $S \preceq^\varepsilon S'$, if there exists an ε -approximate simulation relation from S to S' .

An ε -approximate simulation relation $R \subseteq X \times X'$ is an ε -approximate bisimulation relation between S and S' if R is an ε -approximate simulation relation from S to S' and R^{-1} is an ε -approximate simulation relation from S' to S .

System S and S' are ε -approximate bisimilar, denoted by $S \cong^\varepsilon S'$, if there exists an ε -approximate bisimulation relation R between S and S' .

When $\varepsilon = 0$, the condition (ii) above becomes $(x, x') \in R$ if and only if $H(x) = H'(x')$, and R becomes an exact simulation relation [15]. Similarly, a 0-approximate bisimulation relation R is an exact bisimulation relation.

We next consider constructing *countable* abstractions that are ε -approximate bisimilar to the time discretization of Σ for a given parameter $\varepsilon \in \mathbb{R}^+$.

3 Symbolic Models for Control Systems

3.1 Main Construction

This section contains the main contribution of the paper. We show that a δ -ISS digital control system Σ admits a countable symbolic abstraction. First, we consider a variant of the metric system $S(\Sigma)$ which relates two states if the second is reached from the first in $k \cdot \tau$ time for a parameter $k \in \mathbb{N}$.

Given a constant $k \in \mathbb{N}$ and a digital control system $\Sigma = (\mathbb{R}^n, \tau, \mathcal{U}, \mathcal{U}_\tau, f)$, define the metric system $S_{k\tau}(\Sigma) = (\mathbb{R}^n, \mathcal{U}_\tau, \rightarrow, (\mathbb{R}^n, \|\cdot\|), \lambda x.x)$ where $x \xrightarrow{u} x'$ if there exists a trajectory $\xi_{xu} : [0, k\tau] \rightarrow \mathbb{R}^n$ of Σ satisfying $\xi_{xu}(k\tau) = x'$. Although the metric system $S_{k\tau}(\Sigma)$ relates states of Σ that are k sampling steps apart, this system is not less accurate than $S(\Sigma)$ relating states of Σ one sampling time apart in the sense that for any initialized run with k transitions in the latter there is one transition in the former and vice versa.

Now, assume that Σ is δ -ISS. Consider a triple $\mathbf{q} = (\eta, \mu, k)$, where $\eta \in \mathbb{R}^+$ is the *state space quantization* which determines a discretization of the state space, $\mu \in \mathbb{R}^+$ is the *input set quantization* which determines a discretization of the inputs, and $k \in \mathbb{N}$ is a design parameter. Given Σ and \mathbf{q} , consider the following metric system:

$$S_{\mathbf{q}}(\Sigma) = \left([\mathbb{R}^n]_\eta, \prod_{i=1}^k [\mathcal{U}]_\mu, \rightarrow, (\mathbb{R}^n, \|\cdot\|), H' \right), \quad (5)$$

where $x \xrightarrow{u} x'$ if there is a trajectory $\xi_{xu} : [0, k\tau] \rightarrow \mathbb{R}^n$ such that $\|\xi_{xu}(k\tau) - x'\| \leq \eta/2$ and $H' : [\mathbb{R}^n]_\eta \rightarrow \mathbb{R}^n$ is the natural inclusion map mapping $x \in [\mathbb{R}^n]_\eta$ to itself. We have abused notation by identifying $v \in [\mathcal{U}]_\mu$ with the constant input curve with domain $[0, \tau]$ and value v and identifying $u \in \prod_{i=1}^k [\mathcal{U}]_\mu$ with the

concatenation of k control inputs $v^i \in [\mathbf{U}]_\mu$ (i.e., $u(t) = v^i$ for any $t \in [(i-1)\tau, i\tau]$) for $i = 1, \dots, k$. The set of states of $S_{\mathbf{q}}(\Sigma)$ is countable in general, and finite when the set of states of Σ is restricted to a compact set.

The transition relation of $S_{\mathbf{q}}(\Sigma)$ is well defined in the sense that for every $x \in [\mathbb{R}^n]_\eta$ and every $u \in \prod_{i=1}^k [\mathbf{U}]_\mu$ there always exists $x' \in [\mathbb{R}^n]_\eta$ such that $x \xrightarrow{u} x'$. This can be seen by noting that by definition of $[\mathbb{R}^n]_\eta$, for any $\hat{x} \in \mathbb{R}^n$ there always exists a state $\hat{x}' \in [\mathbb{R}^n]_\eta$ such that $\|\hat{x} - \hat{x}'\| \leq \eta/2$. Hence, for $\xi_{xu}(k\tau)$ there always exists a state $x' \in [\mathbb{R}^n]_\eta$ satisfying $\|\xi_{xu}(k\tau) - x'\| \leq \eta/2$.

We can now present the main result of the paper.

Theorem 1. *Let $\Sigma = (\mathbb{R}^n, \tau, \mathbf{U}, \mathcal{U}_\tau, f)$ be a δ -ISS digital control system, and let functions β and γ satisfy (2). For any $\varepsilon \in \mathbb{R}^+$ and any triple $\mathbf{q} = (\eta, \mu, k)$ of quantization parameters, we have $S_{\mathbf{q}}(\Sigma) \cong^\varepsilon S_{k\tau}(\Sigma)$ if $\mu \leq \text{span}(\mathbf{U})$ and*

$$\beta(\varepsilon, k\tau) + \gamma(\mu) + \eta/2 \leq \varepsilon. \quad (6)$$

Before giving the proof, we point out that if Σ is δ -ISS, there always exists a triple $\mathbf{q} = (\eta, \mu, k)$ satisfying condition (6). Since β is a \mathcal{KL} function, there exists sufficiently large $k \in \mathbb{N}$ such that $\beta(\varepsilon, k\tau) < \varepsilon/2$; for this value of k , by choosing sufficiently small values of η and μ , condition (6) can be fulfilled.

Proof. For notational simplicity, fix $S_{k\tau}(\Sigma) = (X_{k\tau}, U_{k\tau}, \rightarrow_{k\tau}, (\mathbb{R}^n, \|\cdot\|), H_{k\tau})$ and $S_{\mathbf{q}}(\Sigma) = (X_{\mathbf{q}}, U_{\mathbf{q}}, \rightarrow_{\mathbf{q}}, (\mathbb{R}^n, \|\cdot\|), H_{\mathbf{q}})$.

We start by proving $S_{k\tau}(\Sigma) \preceq^\varepsilon S_{\mathbf{q}}(\Sigma)$. Consider the relation $R \subseteq X_{k\tau} \times X_{\mathbf{q}}$ defined by $(x_{k\tau}, x_{\mathbf{q}}) \in R$ if and only if $\|H_{k\tau}(x_{k\tau}) - H_{\mathbf{q}}(x_{\mathbf{q}})\| = \|x_{k\tau} - x_{\mathbf{q}}\| \leq \varepsilon$. Since $X_{k\tau} \subseteq \bigcup_{p \in [\mathbb{R}^n]_\eta} \mathcal{B}_{\eta/2}(p)$ and by (6), for every $x_{k\tau} \in X_{k\tau}$ there always exists $x_{\mathbf{q}} \in X_{\mathbf{q}}$ such that:

$$\|x_{k\tau} - x_{\mathbf{q}}\| \leq \eta/2 \leq \varepsilon. \quad (7)$$

Hence, $(x_{k\tau}, x_{\mathbf{q}}) \in R$ and condition (i) in Definition 4 is satisfied. Now consider any $(x_{k\tau}, x_{\mathbf{q}}) \in R$. Condition (ii) in Definition 4 is satisfied by the definition of R . Let us now show that condition (iii) in Definition 4 holds.

Consider any $v_{k\tau} \in U_{k\tau}$ of duration $k\tau$. Choose an input $u_{\mathbf{q}} \in U_{\mathbf{q}}$ satisfying:

$$\|v_{k\tau}|_{[(l-1)\tau, l\tau]} - u_{\mathbf{q}}|_{[(l-1)\tau, l\tau]}\|_\infty = \|v_{k\tau}((l-1)\tau) - u_{\mathbf{q}}((l-1)\tau)\| \leq \mu, \quad (8)$$

for any $l = 1, \dots, k$. Note that the existence of such $u_{\mathbf{q}}$ is guaranteed by the special shape of \mathbf{U} , described in Definition 1, and by the inequality $\mu \leq \text{span}(\mathbf{U})$ which guarantees that $\mathbf{U} \subseteq \bigcup_{p \in [\mathbf{U}]_\mu} \mathcal{B}_\mu(p)$. Now, we have:

$$\|v_{k\tau} - u_{\mathbf{q}}\|_\infty = \max_{l=1, \dots, k} \left\| v_{k\tau}|_{[(l-1)\tau, l\tau]} - u_{\mathbf{q}}|_{[(l-1)\tau, l\tau]}\right\|_\infty \leq \mu. \quad (9)$$

Consider the unique transition $x_{k\tau} \xrightarrow{v_{k\tau}}_{k\tau} x'_{k\tau} = \xi_{x_{k\tau} v_{k\tau}}(k\tau)$ in $S_{k\tau}(\Sigma)$. It follows from the δ -ISS assumption on Σ and (9) that the distance between $x'_{k\tau}$ and $\xi_{x_{\mathbf{q}} u_{\mathbf{q}}}(k\tau)$ is bounded as:

$$\|x'_{k\tau} - \xi_{x_{\mathbf{q}} u_{\mathbf{q}}}(k\tau)\| \leq \beta(\|x_{k\tau} - x_{\mathbf{q}}\|, k\tau) + \gamma(\|v_{k\tau} - u_{\mathbf{q}}\|_\infty) \leq \beta(\varepsilon, k\tau) + \gamma(\mu). \quad (10)$$

Since $X_{k\tau} \subseteq \bigcup_{p \in [\mathbb{R}^n]_n} \mathcal{B}_{\eta/2}(p)$, there exists $x'_q \in X_q$ such that:

$$\|x_{x_q u_q}(k\tau) - x'_q\| \leq \eta/2, \quad (11)$$

which, by the definition of $S_q(\Sigma)$, implies the existence of $x_q \xrightarrow{u_q} q x'_q$ in $S_q(\Sigma)$. Using the inequalities (6), (10), (11), and triangle inequality, we obtain:

$$\begin{aligned} \|x'_{k\tau} - x'_q\| &\leq \|x'_{k\tau} - \xi_{x_q u_q}(k\tau) + \xi_{x_q u_q}(k\tau) - x'_q\| \\ &\leq \|x'_{k\tau} - \xi_{x_q u_q}(k\tau)\| + \|\xi_{x_q u_q}(k\tau) - x'_q\| \\ &\leq \beta(\varepsilon, k\tau) + \gamma(\mu) + \eta/2 \leq \varepsilon. \end{aligned}$$

Therefore, we conclude $(x'_{k\tau}, x'_q) \in R$ and condition (iii) in Definition 4 holds.

Now we prove $S_q(\Sigma) \preceq^\varepsilon S_{k\tau}(\Sigma)$ implying that R^{-1} is a suitable ε -approximate simulation relation. Consider the relation $R \subseteq X_{k\tau} \times X_q$, defined in the first part of the proof. For every $x_q \in X_q$, by choosing $x_{k\tau} = x_q$, we have $(x_{k\tau}, x_q) \in R$ and condition (i) in Definition 4 is satisfied. Now consider any $(x_{k\tau}, x_q) \in R$. Condition (ii) in Definition 4 is satisfied by the definition of R . Let us now show that condition (iii) in Definition 4 holds. Consider any $u_q \in U_q$. Choose the input $v_{k\tau} = u_q$ and consider the unique $x'_{k\tau} = \xi_{x_{k\tau} v_{k\tau}}(k\tau)$ in $S_{k\tau}(\Sigma)$. Using δ -ISS assumption for Σ , we bound the distance between $x'_{k\tau}$ and $\xi_{x_q u_q}(k\tau)$ as:

$$\|x'_{k\tau} - \xi_{x_q u_q}(k\tau)\| \leq \beta(\|x_{k\tau} - x_q\|, k\tau) \leq \beta(\varepsilon, k\tau). \quad (12)$$

Using the definition of $S_q(\Sigma)$, the inequalities (6), (12), and the triangle inequality, we obtain:

$$\begin{aligned} \|x'_{k\tau} - x'_q\| &\leq \|x'_{k\tau} - \xi_{x_q u_q}(k\tau) + \xi_{x_q u_q}(k\tau) - x'_q\| \\ &\leq \|x'_{k\tau} - \xi_{x_q u_q}(k\tau)\| + \|\xi_{x_q u_q}(k\tau) - x'_q\| \\ &\leq \beta(\varepsilon, k\tau) + \eta/2 \leq \varepsilon. \end{aligned}$$

Therefore, we conclude that $(x'_{k\tau}, x'_q) \in R$ and condition (iii) in Definition 4 holds. \square

Remark 2. Although we assume the set U is infinite, Theorem 4 still holds when the set U is finite, with the following modifications. First, the system Σ is required to satisfy the property (2) for $v = v'$. Second, take $U_q = \prod_{i=1}^k U$ in the definition of $S_q(\Sigma)$. Finally, in the condition (6), set $\mu = 0$.

A concern that arises when using $S_q(\Sigma)$ is the inter-samples behavior: can a specification be violated for $t \in]0, k\tau]$ even though it is satisfied at $t = 0$ and $t = k\tau$? This concern arises already in existing approaches to compute discrete abstractions [17][18][10] (setting $k = 1$).

In the absence of any bounds on inter-samples behaviors, the results of controller synthesis on $S_q(\Sigma)$ can be interpreted in the following way. If there is no controller satisfying a safety or co-Büchi specification on $S_q(\Sigma)$, respectively, then we can conclude that there is no controller satisfying the same safety or

co-Büchi specification on Σ , respectively. Dually, if there is a controller satisfying a reachability or Büchi specification on $S_q(\Sigma)$, respectively, then we can conclude that the refinement of that controller satisfies the same reachability or Büchi specification on Σ , respectively.

In practice, the parameter τ is chosen to be sufficiently small, and if $k \in \mathbb{N}$ is also small, the specification is directly verified against $S_q(\Sigma)$ ignoring inter-samples behaviors. If it is important to include the effects of inter-samples behaviors, e.g., when τ or k are large, there is a naive way to solve the inter-samples behaviors, especially in terms of synthesizing a controller. In the process of constructing abstract transition system $S_q(\Sigma)$, every transition can be labeled not only with the input but also with the sequence of the states visited at times $\tau, 2\tau, \dots, (k-1)\tau$. Now, one can find a symbolic controller for the constructed abstract transition system with the knowledge of what is happening in the inert-samples. By doing this, we shrink the inter-samples behaviors in only one sample time $[0, \tau]$.

Furthermore, one can over-approximate the reachable states between two sample points using techniques incorporating zonotopes [7, 20] or support functions [11, 6]. We illustrate the bounding technique using zonotopes. A transition $x_q \xrightarrow{u_q} q x'_q$ in $S_q(\Sigma)$ implies the existence of a trajectory $\xi_{x_q u_q}$ of Σ satisfying $\|\xi_{x_q u_q}(k\tau) - x'_q\| \leq \eta/2$. We can thus enclose x_q in a zonotope Z_1 , enclose $\mathcal{B}_{\eta/2}(x'_q)$ in a different zonotope Z_2 , and use results in [7] (see also Proposition 7.31 in [20]) for a given $u_q \in U_q$ to obtain another zonotope $Z_{k\tau}(x_q, u_q, x'_q)$ containing all the states $\xi_{x_q u_q}(t)$ for $t \in [0, k\tau]$.

Fix an ε and q such that $S_q(\Sigma) \cong^\varepsilon S_{k\tau}(\Sigma)$. Let Z be the smallest zonotope enclosing Z_1 and Z_2 . Let $\varepsilon_0(x_q, u_q, x'_q)$ be an upper bound on the Hausdorff distance between Z and $Z_{k\tau}(x_q, u_q, x'_q)$, and let ε_0 be the supremum over all choices of $x_q \xrightarrow{u_q} q x'_q$. Then, if $S_q(\Sigma) \cong^{\varepsilon-\varepsilon_0} S_{k\tau}(\Sigma)$, then we know that any trajectory of $S_q(\Sigma)$ is at most ε away from a trajectory of $S(\Sigma)$. If $\varepsilon_0 > \varepsilon$, then one needs to reduce the original precision ε and compute a new q , and iterate. This represents the tradeoff between choosing larger k 's and bounding the deviations of inter-samples behaviors: choosing a larger k makes satisfying (6) easier, but can make ε_0 larger.

Remark 3. For linear control systems and safety (or co-Büchi) specifications, we can compute bounds on inter-samples behaviors in the following way. Assume $W \in \mathbb{R}^n$ is a compact and convex polyhedron with h vertices x^1, \dots, x^h . Assume Σ is a globally asymptotically stable linear control system, defined in (3), with a compact input set U and $S_q(\Sigma)$ is its symbolic abstraction. Assume there exists a symbolic controller on $S_q(\Sigma)$ satisfying¹ $\square W$. What can we say about the existence of a controller on $S(\Sigma)$? It can be readily verified that there exists a controller satisfying $\square \widehat{W}$ on $S(\Sigma)$, where \widehat{W} is the polyhedron with vertices $\widehat{x}^1, \dots, \widehat{x}^k$, defined by:

$$\widehat{x}^i = e^{Al^*\tau} x^i + A^{-1} (e^{A\tau} - I_n) \left[e^{A(l^*-1)\tau} Bu_1^* + \dots + e^{A\tau} Bu_{l^*-1}^* + Bu_{l^*}^* \right], \quad (13)$$

¹ Note that the semantics of LTL would be defined over the output behaviors of $S_q(\Sigma)$.

where l^* and u^* are computed by:

$$(l^*, u_1^*, \dots, u_{l^*}^*) = \arg \max_{l=1, \dots, k-1} \min_{u_1, \dots, u_l \in U} \|\tilde{x}^i\|_W, \quad (14)$$

where the symbol $\|\cdot\|_W$ denotes the point-to-set distance, *viz.* $\|x\|_W = \min_{w \in W} \|x - w\|$, and

$$\tilde{x}^i = e^{Al\tau} x^i + A^{-1} (e^{A\tau} - I_n) \left[e^{A(l-1)\tau} Bu_1 + \dots + e^{A\tau} Bu_{l-1} + Bu_l \right]. \quad (15)$$

If $\widehat{W} \subseteq W$, then no new states are introduced through inter-samples behaviors. A similar analysis can be performed for co-Büchi objectives.

3.2 Comparison with Previous Techniques

We now compare our result (Theorem 1) with existing results on computing ε -approximate bisimilar discrete abstractions for δ -ISS digital control systems.

The construction in Pola, Girard, and Tabuada [17] essentially fixes $k = 1$. That is, it computes the metric system $S(\Sigma)$ and shows that $S_q(\Sigma)$ is ε -approximate bisimilar to it if $\mu \leq \text{span}(U)$ and $\beta(\varepsilon, \tau) + \gamma(\mu) + \eta/2 \leq \varepsilon$. This inequality may not hold for a choice of τ and in that case, the technique fails to construct an ε -approximate bisimilar abstraction. In contrast, we are guaranteed that for every given ε and τ , we can choose parameters η , μ , and k such that $S_q(\Sigma)$ is ε -approximate bisimilar to $S_{k\tau}(\Sigma)$.

Next, we compare with the construction in Girard, Pola, and Tabuada [10]. First, we need the notion of δ -ISS Lyapunov functions.

Definition 5. [3] Fix a control system Σ . A smooth function $V : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}_0^+$ is called a δ -ISS Lyapunov function for Σ if there exist \mathcal{K}_∞ functions $\underline{\alpha}$, $\overline{\alpha}$, and σ , and a positive real $\kappa \in \mathbb{R}^+$ such that:

- (i) for any $x, x' \in \mathbb{R}^n$, $\underline{\alpha}(\|x - x'\|) \leq V(x, x') \leq \overline{\alpha}(\|x - x'\|)$;
- (ii) for any $x, x' \in \mathbb{R}^n$ and for any $u, u' \in U$,

$$\frac{\partial V}{\partial x} f(x, u) + \frac{\partial V}{\partial x'} f(x', u') \leq -\kappa V(x, x') + \sigma(\|u - u'\|).$$

The following result characterizes δ -ISS in terms of δ -ISS Lyapunov functions.

Theorem 2. [3] Consider the digital control system $\Sigma = (\mathbb{R}^n, \tau, U, \mathcal{U}_\tau, f)$. If U is convex, compact, contains the origin, and $f(0, 0) = 0$, then Σ is δ -ISS iff it admits a δ -ISS Lyapunov function.

The results in [10] additionally assume:

$$|V(x, y) - V(x, z)| \leq \hat{\gamma}(\|y - z\|), \quad (16)$$

for any $x, y, z \in \mathbb{R}^n$, and some \mathcal{K}_∞ function $\hat{\gamma}$. As explained in [10], this assumption is not restrictive provided V is smooth and we are interested in the dynamics of Σ on a compact subset of \mathbb{R}^n , which is often the case in practice. The main result of [10] is as follows.

Table 1. Parameters of $S_q(\Sigma)$ and overall required time for constructing $S_q(\Sigma)$ and synthesizing controllers. The notation N/A means not applicable. We use the notation ∞ to indicate that the size of $S_q(\Sigma)$ is too large for *Pessoa* to finish constructing the abstraction.

Control Systems	τ	ε	Parameters of $S_q(\Sigma)$			Time		
			$\mu=0.5$			[17]	[10]	Our approach
DC motor	0.02	1	N/A	0.012	(0.5, 2)	N/A	∞	1.42s
Robot	0.002	0.075	0.0027	0.0022	(0.01, 4)	∞	∞	45.29s
Pendulum	0.02	0.25	N/A	0.0007	(0.1, 2)	N/A	∞	33s

Theorem 3. [10] Let Σ be a δ -ISS digital control system admitting a δ -ISS Lyapunov function V . For any $\varepsilon \in \mathbb{R}^+$ and parameters $q = (\eta, \mu, 1)$, we have $S_q(\Sigma) \cong^\varepsilon S(\Sigma)$ if $\mu \leq \text{span}(U)$ and

$$\eta/2 \leq \min \left\{ \hat{\gamma}^{-1} \left((1 - e^{-\kappa\tau}) \underline{\alpha}(\varepsilon) - \frac{1}{\kappa} \sigma(\mu) \right), \bar{\alpha}^{-1}(\underline{\alpha}(\varepsilon)) \right\}, \quad (17)$$

For a given sampling time $\tau \in \mathbb{R}^+$, there always exist $\eta, \mu \in \mathbb{R}^+$ satisfying the condition (17). However, it can be readily verified that if the sampling time τ is very small, the right hand side of the inequality (17) is very small as well. Therefore, the upper bound on η will be very small, resulting in a large symbolic abstraction. On the other hand, we can always choose $k \in \mathbb{N}$ in (6) appropriately, to control the size of the symbolic model, justifying advantage of our proposed approach in comparison with the approach in [10]. In the next section, we demonstrate experimentally that our approach can result in discrete abstractions with orders of magnitude fewer states than the abstractions using Theorem 3.

4 Examples

We now experimentally demonstrate the effectiveness of our new construction. In the examples below, the computation of the abstractions $S_q(\Sigma)$ was performed using the tool *Pessoa* [16] on a laptop with CPU Intel Core 2 Duo @ 2.4GHz. We assume that control inputs are piecewise constant of duration τ and that U_τ is finite and contains curves taking values in $[U]_{0.5}$. Hence, as explained in Remark 2, $\mu = 0$ in the conditions (6) and (17). In the examples below, all constants and variables use SI units. Controllers enforcing the specifications were found by using standard algorithms from game theory, see e.g. [14, 20], as implemented in *Pessoa*. Table II summarizes the experimental results.

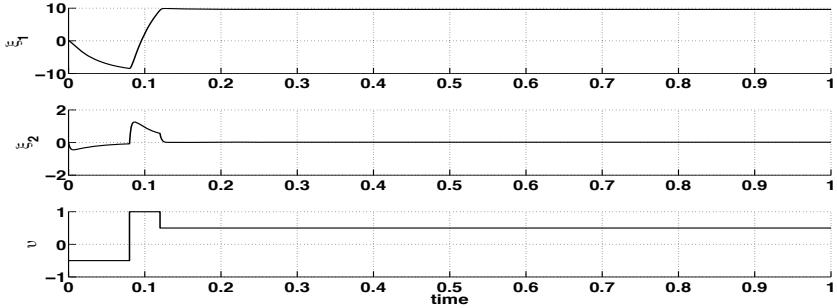


Fig. 1. Upper and central panels: evolution of ξ_1 and ξ_2 with initial condition $(0, 0)$. Lower panel: input signal.

4.1 DC Motor

Model: Consider a linear DC motor (from [1]) described by:

$$\Sigma : \begin{cases} \dot{\xi}_1 = -\frac{b}{J}\xi_1 + \frac{K}{J}\xi_2, \\ \dot{\xi}_2 = -\frac{K}{L}\xi_1 - \frac{R}{L}\xi_2 + \frac{1}{L}v, \end{cases} \quad (18)$$

where ξ_1 is the angular velocity of the motor, ξ_2 is the current through the inductor, v is the source voltage, $b = 10^{-4}$ is the damping ratio of the mechanical system, $J = 10^{-4}$ is the moment of inertia of the rotor, $K = 5 \times 10^{-2}$ is the electromotive force constant, $L = 2 \times 10^{-3}$ is the electric inductance, and $R = 1$ is the electric resistance. Using Remark 1, it is readily seen that Σ is δ -ISS.

Abstraction: We assume that $U = [-1, 1]$. We work on the subset $D = [-10, 10] \times [-10, 10]$ of the state space of Σ . For a sampling time $\tau = 0.02$, the function β in (1) is given by $\beta(\varepsilon, \tau) = 1.26\varepsilon$. Hence, the results in [17] cannot be applied because the condition (6) of Theorem 1 cannot be fulfilled when $k = 1$. On the other hand, by choosing $k = 2$, we have $\beta(\varepsilon, k\tau) = 0.73\varepsilon$ implying that the condition (6) of Theorem 1 can be fulfilled. For a precision $\varepsilon = 1$, we construct a symbolic model $S_q(\Sigma)$. The parameters of $S_q(\Sigma)$ based on the results in this paper as well as the construction in [10] are given in Table 1. The proposed state space quantization parameter in [10] is roughly 42 times smaller than our quantization parameter. Since Σ is a 2 dimensional system, the size of our abstraction is 42^2 times smaller than the one in [10].

Example control problem: Consider the objective to design a controller forcing the trajectories of Σ to reach and stay within $W = [9, 10] \times [-1, 1]$ thereafter while always remaining within $Z = [-10, 10] \times [-1, 1]$, that is, the LTL specification $\diamond \square W \wedge \square Z$. Using the result in Remark 3, we compute \widehat{Z} and \widehat{W} and note that in this case, $\widehat{Z} \subseteq Z$ and $\widehat{W} \subseteq W$. Hence, the existence of a symbolic controller on $S_q(\Sigma)$ satisfying $\diamond \square W \wedge \square Z$ implies the existence of a controller satisfying $\diamond \square W \wedge \square Z$ on Σ . In Figure 1, we show the closed-loop trajectory stemming from the initial condition $(0, 0)$ as well as the evolution of the input signal. It

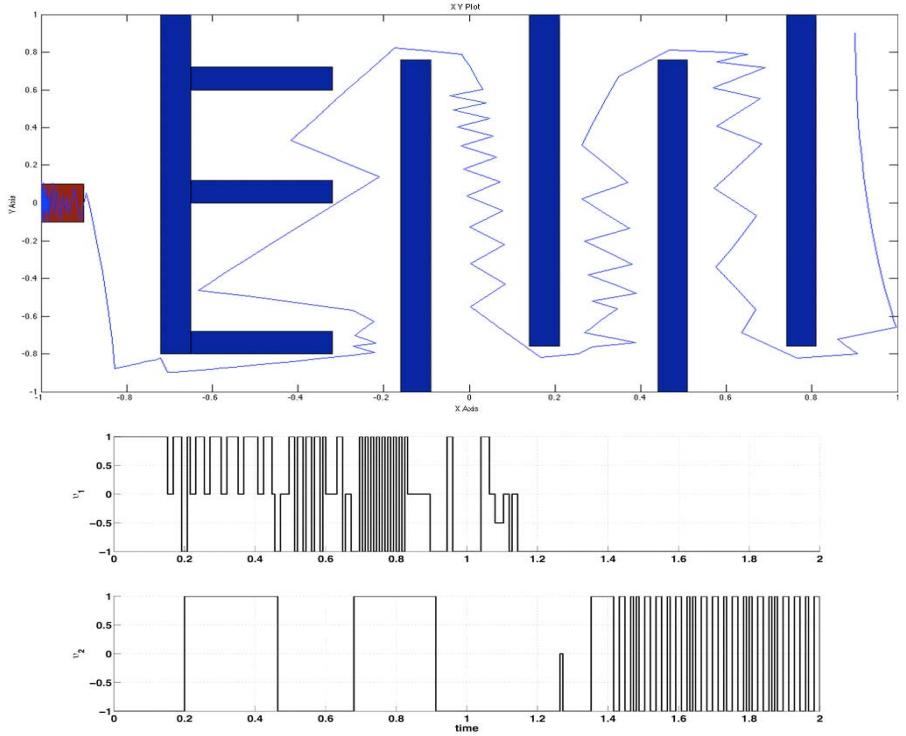


Fig. 2. Evolution of the robot and the input signals with initial condition (0.9, 0.9)

is readily seen that the specifications are satisfied in the sense that trajectories of Σ reach and stay within $W^\varepsilon = [9 - \varepsilon, 10] \times [-1 - \varepsilon, 1 + \varepsilon]$ thereafter while always remaining within $Z^\varepsilon = [-10, 10] \times [-1 - \varepsilon, 1 + \varepsilon]$.

4.2 Motion Planing

Model: Consider a linear model of a robot described by:

$$\Sigma : \begin{cases} \dot{\xi}_1 = -10\xi_1 - \xi_2 + 10v_1, \\ \dot{\xi}_2 = -\xi_1 - 10\xi_2 + 10v_2. \end{cases} \quad (19)$$

The position of the robot is given by the pair (ξ_1, ξ_2) . The pair (v_1, v_2) are the control inputs, expressing the velocity of the wheels. Using Remark 1, it is readily seen that Σ is δ -ISS.

Abstraction: We assume that $(v_1, v_2) \in U = [-1, 1] \times [-1, 1]$. We work on the subset $D = [-1, 1] \times [-1, 1]$ of the state space of Σ . For a sampling time $\tau = 0.002$, the function β in (4) is given by $\beta(\varepsilon, \tau) = 0.982\varepsilon$. Hence, the results in [17] can be applied. On the other hand, by choosing $k = 4$, we have $\beta(\varepsilon, k\tau) = 0.93\varepsilon$ implying that the condition (6) of Theorem 1 can also be fulfilled. For a

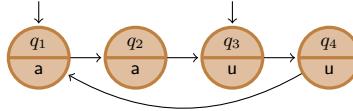


Fig. 3. Finite system describing the schedulability constraints. The lower part of the states are labeled with the outputs a and u denoting availability and unavailability of the microprocessor, respectively.

precision $\varepsilon = 0.075$, we construct a symbolic model $S_q(\Sigma)$. The parameters of $S_q(\Sigma)$ based on the results in this paper and those from [17][10] are given in Table II. The state space quantization parameters in [17][10] are roughly four times smaller than our η . Therefore, the size of our abstraction is roughly 4^2 times smaller than the ones in [17][10].

Example control problem: Consider the problem of designing a controller navigating the robot to reach the target set $W = [-1, -0.9] \times [-0.1, 0.1]$, indicated with a target box in the far left hand side in Figure 2, while avoiding the obstacles, indicated as rectangular boxes in Figure 2, and then remain indefinitely inside W . If we denote by ϕ and ψ the predicates representing the target and obstacles, respectively, this specification can also be expressed by the LTL formula $\diamond\Box\phi \wedge \Box\neg\psi$. If we express the non-obstacle area in Figure 2 as the union of l polyhedra Z_i , for $i = 1, \dots, l$, then using the result in Remark 3, we compute \widehat{Z}_i and \widehat{W} , and note that for this example, $\widehat{Z}_i \subseteq Z_i$, for each $i = 1, \dots, l$, and $\widehat{W} \subseteq W$. Hence, a symbolic controller on $S_q(\Sigma)$ satisfying $\diamond\Box\phi \wedge \Box\neg\psi$ implies there exists a controller satisfying the specification on Σ . In Figure 2, we show the closed-loop trajectory stemming from the initial condition $(0.9, 0.9)$ and the evolution of the input signals. It is readily seen that the specification is satisfied.

4.3 Pendulum with Resource Constraints

Model: Consider a nonlinear model of a pendulum on a cart (from [17]) described by:

$$\Sigma : \begin{cases} \dot{\xi}_1 = \xi_2, \\ \dot{\xi}_2 = -\frac{g}{l} \sin(\xi_1) - \frac{h}{m} \xi_2 + v, \end{cases} \quad (20)$$

where ξ_1 and ξ_2 are the angular position and velocity of the point mass, v is the torque applied to the cart, $g = 9.8$ is acceleration due to gravity, $l = 5$ is the length of the rod, $m = 0.5$ is the mass, and $h = 3$ is the coefficient of friction. As shown in [17], Σ is δ -ISS.

Abstraction: We assume that $U = [-1.5, 1.5]$. We work on the subset $D = [-1, 1] \times [-1, 1]$ of the state space of Σ . As shown in [17], the function β in (2) is given by $\beta(\varepsilon, \tau) = 6.17e^{-2.08\tau}\varepsilon$, so for a sampling time $\tau = 0.5$, we have $\beta(\varepsilon, \tau) = 2.18\varepsilon$. Hence, the results in [17] cannot be applied because the condition (6) of Theorem II cannot be fulfilled when $k = 1$. On the other hand,

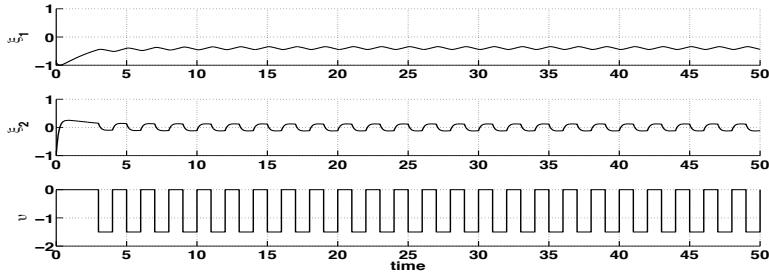


Fig. 4. Upper and central panels: evolution of ξ_1 and ξ_2 with initial condition $(-0.9, -1)$. Lower panel: input signal.

by choosing $k = 2$, we have $\beta(\varepsilon, k\tau) = 0.77\varepsilon$, so the condition (6) of Theorem 1 is fulfilled. For a precision $\varepsilon = 0.25$, we construct a symbolic model $S_q(\Sigma)$. The parameters of $S_q(\Sigma)$ based on the results in this paper and [10] are given in Table 1. The state space quantization parameter in [10] is roughly 147 times smaller than our quantization parameter. Therefore, the size of the symbolic model computed by our algorithm is roughly $147^2 \sim 2 \times 10^4$ times smaller than the one in [10].

Example control problem: Suppose our objective is to design a controller forcing the trajectories of the system to reach the target set $W = [-0.7, -0.6] \times [-1, 1]$ and to remain indefinitely inside W . Furthermore, to add a discrete component to the problem, we assume that the controller is implemented on a microprocessor, executing other tasks in addition to the control task. We consider a schedule with epochs of four time slots in which the first two slots are allocated to the control task and the rest of them to other tasks. The expression time slot refer to a time interval of the form $[k'\tau, (k' + 1)\tau[$ with $k' \in \mathbb{N}$ and where τ is the sampling time. Therefore, the microprocessor schedule is given by:

$$|aauu|aauu|aauu|aauu|aauu|aauu|aauu|\dots,$$

where **a** denotes a slot available for the control task and **u** denotes a slot allotted to other tasks. The symbol | separates each epoch of four time slots. The schedulability constraint on the microprocessor can be represented by the finite system in Figure 3. Initial states of the finite system are distinguished by being the target of a sourceless arrow. In Figure 4, we show the closed-loop trajectory stemming from the initial condition $(-0.9, -1)$, and the evolution of the input signal, where the finite system initialized from state q_3 .

References

1. Control tutorial for Matlab and Simulink. Electronically available at, <http://www.library.cmu.edu/ctms/ctms/>

2. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* 126, 183–235 (1994)
3. Angeli, D.: A Lyapunov approach to incremental stability properties. *IEEE Transactions on Automatic Control* 47(3), 410–421 (2002)
4. Angeli, D., Sontag, E.D.: Forward completeness, unboundedness observability, and their Lyapunov characterizations. *Systems and Control Letters* 38, 209–217 (1999)
5. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: FOCS 1991: Foundations of Computer Science, pp. 368–377. IEEE (1991)
6. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
7. Girard, A.: Reachability of Uncertain Linear Systems Using Zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005)
8. Girard, A.: Approximately Bisimilar Finite Abstractions of Stable Linear Systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 231–244. Springer, Heidelberg (2007)
9. Girard, A., Pappas, G.J.: Approximation metrics for discrete and continuous systems. *IEEE Transactions on Automatic Control* 25(5), 782–798 (2007)
10. Girard, A., Pola, G., Tabuada, P.: Approximately bisimilar symbolic models for incrementally stable switched systems. *IEEE Transactions on Automatic Control* 55(1), 116–126 (2009)
11. Le Guernic, C., Girard, A.: Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems* 4(2), 250–262 (2010)
12. Henzinger, T.A.: Hybrid Automata with Finite Bisimulations. In: Fülöp, Z. (ed.) ICALP 1995. LNCS, vol. 944, pp. 324–335. Springer, Heidelberg (1995)
13. Kailath, T.: Linear systems. Prentice-Hall, Inc. (1980)
14. Maler, O., Pnueli, A., Sifakis, J.: On the Synthesis of Discrete Controllers for Timed Systems. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 229–242. Springer, Heidelberg (1995)
15. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc. (1989)
16. Pessoa (2009), <http://www.cyphylab.ee.ucla.edu/pessoa>
17. Pola, G., Girard, A., Tabuada, P.: Approximately bisimilar symbolic models for nonlinear control systems. *Automatica* 44(10), 2508–2516 (2008)
18. Pola, G., Tabuada, P.: Symbolic models for nonlinear control systems: alternating approximate bisimulations. *SIAM Journal on Control and Optimization* 48(2), 719–733 (2009)
19. Sontag, E.D.: Mathematical control theory, 2nd edn. Springer (1998)
20. Tabuada, P.: Verification and Control of Hybrid Systems, A symbolic approach. Springer, US (2009)
21. Teel, A.R., Nesic, D., Kokotovic, P.V.: A note on input-to-state stability of sampled-data nonlinear systems. In: CDC 1998: Conference on Decision and Control, pp. 2473–2478. IEEE (1998)
22. Thomas, W.: On the Synthesis of Strategies in Infinite Games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
23. Zamani, M., Tabuada, P.: Backstepping design for incremental stability. *IEEE Transaction on Automatic Control* 56(9), 2184–2189 (2011)

Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System

Alessandro Cimatti¹, Raffaele Corvino², Armando Lazzaro², Iman Narasamdy¹, Tiziana Rizzo², Marco Roveri¹, Angela Sanseviero², and Andrei Tchaltsev¹

¹ Fondazione Bruno Kessler — IRST

² Ansaldo-STS

Abstract. Formal verification and validation is a fundamental step for the certification of railways critical systems. Many railways safety standards (e.g. the CENELEC EN-50126, EN-50128 and EN-50129 standards implement the mandatory safety requirements of IEC-61508-7 standard for Functional and Safety) currently mandate the use of formal methods in the design to certify correctness.

In this paper we describe an industrial application of formal methods for the verification and validation of “Logica di Sicurezza” (LDS), the safety logic of a railways ERTMS Level 2 system developed by Ansaldo-STS. LDS is a generic control software that needs to be instantiated on a railways network configuration. We developed a methodology for the verification and validation of a critical subset of LDS deployed on typical realistic railways network configurations. To show feasibility, effectiveness and scalability, we have experimented with several state of the art symbolic software model checking techniques and tools on different network configurations. From the experiments, we have successfully identified an effective strategy for the verification and validation of our case study. Moreover, the results of experiments show that formal verification and validation is feasible and effective, and also scales reasonably well with the size of the configuration. Given the results, Ansaldo-STS is currently integrating the methodology in its internal Development and Verification & Validation Flow.

1 Introduction

The verification of industrial safety critical systems is paramount. It is a particularly difficult activity because of the size and complexity of the systems. The most frequently used methods, simulation and testing, can increase the reliability of the systems, but, since they are not exhaustive, they cannot show the absence of errors. Failure in detecting an error in a safety critical system can lead to a catastrophic situation.

Formal verification has proved to provide a complete coverage. Particularly for railways critical systems, formal verification is becoming fundamental for the certification of such systems. The CENELEC EN-50126, EN-50128 and EN-50129 standards, that implement the mandatory safety requirements of the IEC-61508-7 standard for Functional and Safety, require the use of formal verification techniques to certify the correctness of the design. Despite their importance, the application of such techniques in industrial settings is by no means trivial. First, a proper verification and validation methodology has to be designed to allow for an efficient handling of the size of the system under verification. Second, the verification and validation techniques should integrate smoothly in the company internal Development, Verification & Validation Flow.

In this paper we describe an industrial application of formal methods for the verification and validation of a fragment of railways critical software. The software is called *Logica di Sicurezza* (LDS), which is a realization of the safety logic of a Level 2 European Railways Train Management System (ERTMS) developed by Ansaldo-STS. LDS is a generic distributed complex control software, which is designed to manage and control the train spacing in an ERTMS. LDS also guarantees the safety of the controlled railway network, e.g., no train collisions and proper distance among the trains. LDS is programmable and scalable, in the sense that, different train spacing control systems can be developed by instantiating the generic controller to a specific railway network configuration. For our case study, we have focused on radio block sections (RBS), which are a critical fragment of LDS that controls routes in railways. We consider instantiations of RBS on typical realistic railways network configurations.

The verification of the RBS application, carried out by Ansaldo-STS, has been relying on manual inspections and on simulations, where the simulation vectors were manually designed based on the experience of the engineers. Achieving full coverage was considered a very challenging task. We show in this paper a methodology that we have identified for applying formal verification to the considered application. This methodology is a result of a thorough evaluation of the state-of-the-art verification techniques that have shown to be highly effective in the verification of large systems. In particular, we focus on symbolic model checking for software as it is exhaustive and completely automatic, and thus allows for an easy integration within the Development and Verification & Validation Flow.

We have devised a verification flow for the verification of the considered application. We first translate the specifications of the case study, written in the VELOS language, a restricted version of the C++ language developed and used by Ansaldo-STS, into forms that can be verified by existing tools. In particular we translate such specifications into NUSMV models and sequential C programs. The translation into NUSMV models allows us to use the portfolio of advanced verification techniques, like bounded model checking [7], temporal induction [19], and CEGAR [15], that have been implemented in an extended version of the NUSMV model checker [12] developed within Fondazione Bruno Kessler (FBK). The translation into sequential C programs allows us to experiment with advanced software model checking techniques, like the eager and lazy predicate abstractions [17,4], and enables the use of off-the-shelf software model checkers for C, like CBMC [16], BLAST [4], SATABS [17], CPACHECKER [5], and KRATOS [14] (developed by FBK). We have further customized KRATOS and have extended its analysis to achieve the verification needs identified during the project.

We have carried out experiments with the above mentioned verification tools on a significant set of benchmarks obtained from the critical subset of the considered application. The experiment results allow us to devise an effective strategy, based on a combination of verification approaches, that greatly improves the efficiency of the verification. The results also provide evidence that formal verification in our industrial setting is feasible and effective. Given these results, Ansaldo-STS is currently integrating the methodology and tools in its Development and Verification & Validation Flow.

This paper is organized as follows. In Section 2 we describe LDS application. In Section 3 we present an overview of formal verification techniques. In Section 4 we

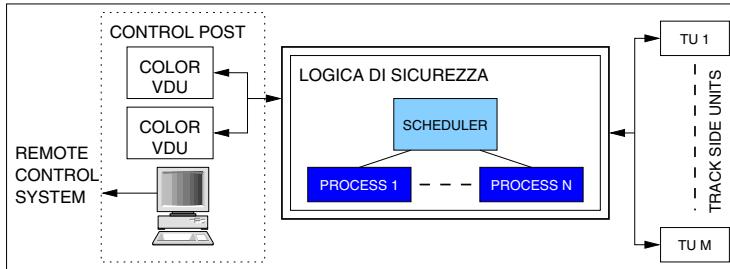


Fig. 1. The application and its environment

describe the verification methodology, along with the extensions to KRATOS. In Section 5 we show the results of our experimental evaluation. In Section 6 we discuss related work. Finally, in Section 7 we draw some conclusion, including the lessons learned, and discuss future work.

2 The Application: *Logica Di Sicurezza*

The work in this paper is concerned with the verification of a complex real-world safety critical application developed by Ansaldo-STS. The application is used to manage and control the train spacing in an ERTMS railway system. Figure 1 shows a high-level architecture of the application and its environment. Our work has focused on *Logica di Sicurezza* (LDS), a software subsystem that controls train movements and track-side equipment that is connected to track-side units, e.g., track circuits, signals and switches. LDS also implements the logical functions that can be requested by human operators, e.g., preparing the tracks for moving trains.

LDS is highly programmable and scalable: it is possible to program the modalities under which the requested logical functions are performed; and to program various configurations of track-side units. Such a distinguishing feature is achieved by means of a logical architecture composed of a *scheduler* controlling the activation of application-dependent processes. LDS is designed by specifying the processes controlled by the scheduler, which are then converted into executable code. In general, LDS can be thought of as a reactive system, acting along the following loop: read status from track-side units and input from the operator, run the processes through the scheduler to apply the control law of the logic, and write commands to actuators of the units.

The specification of processes in this architecture is non-trivial. Indeed, a railway station can have a large number of physical devices, and processes of many different kinds are required to take into account the relations and interconnections among physical devices. Moreover, although the software is completely deterministic and the possible external events (e.g. faults of peripheral devices) have been exhaustively classified, there is still a high degree of non-determinism. The software does not know if and when external events will happen, e.g., certain tasks can be requested at any time. Furthermore, the physical devices will typically react to controls with (unpredictable) delays, and may even show faulty behaviors.

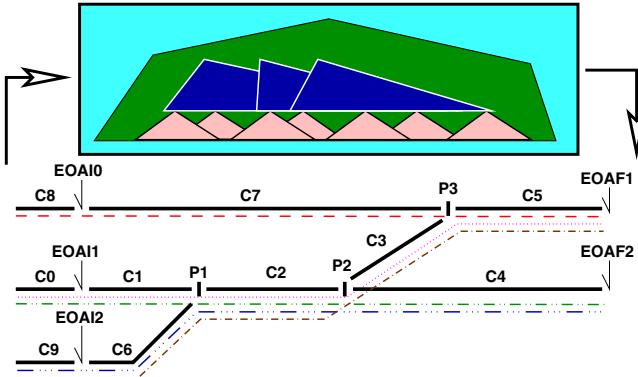


Fig. 2. A radio block section fragment of LDS and the railway layout that the fragment controls

Currently, the specification is validated by means of traditional techniques, such as simulation. Designing test cases that ensure high code coverage, as well as stimulate the whole specification to exercise all the critical functionalities, as mandated by the standards, is non-trivial. This is typically done manually, and thus often miss some critical functionalities.

2.1 The Radio Block Section

In this paper, we focus on the radio block section (RBS) fragment of LDS, a critical subsystem responsible for the allocation of logical routes to trains. In particular, as a case study, we consider the configuration corresponding to the physical layout of the railways depicted in the lower part of Figure 2 with thick dark lines. Some entities in this fragment correspond to physical entities, while some others to logical ones. A *component* is a logical reservation for a segment of line. A *point* is a logical controller for a switch. For example, the component C_i and the point P_i correspond, respectively, to the segment C_i and the switch P_i in Figure 2. A *radio block section* (RBS) corresponds to a logical route through the physical displacement. An RBS is composed of several components and points, and enclosed by an initial and final end-of-authority (EOA). In Figure 2 EOA_{I0}, EOA_{I1}, and EOA_{I2} are the initial EOA, while EOAF₁ and EOAF₂ are the final EOA. Figure 2 shows the ten RBS's of the physical layout in different dashed/dotted (colored) lines; each route in the figure denotes two RBS's, for left-to-right and right-to-left directions.

Figure 2 (upper part) shows the fragment of LDS that we have focused on in this activity. The (light blue) rectangle represents the scheduler, the (green) cone is the whole set of processes in LDS, the dark (blue) triangles represent the RBS's, and the light (pink) triangles represent the call graphs of the “underlying” processes. Each process consists of data and functions to modify the data. Processes are organized hierarchically according to the call graph of the whole logic of LDS, e.g., RBS processes can call the functions of sub-processes illustrated in Figure 2 by the light (pink) triangles.

2.2 Verification Properties of the RBS

Ansaldo-STS has identified five parametric properties for the RBS fragment: one for each component and four for each RBS. These properties correspond to the safety requirements of the considered application, and are formulated as invariants that have to hold at every read-scheduler-output cycle. For the case study in Figure 2 there are 12 components and 10 RBS's, and this amounts to have 52 properties. An example of a property is “no two different trains occupy the same track”. An additional property is introduced to test the consistency of the physical layout configuration, so in total we have 53 properties.

Since the focus of the analysis is on the RBS's, along with the processes that they induce, we abstract away non-RBS processes, and use an abstracted scheduler that repeatedly chooses one of the ten RBS's, depending on a certain condition, starts its process by executing one of its functions, until some exiting condition is met. To avoid having a too coarse abstraction, we specified a number of assumptions, e.g., constraints over signals, that have to be satisfied by the abstracted parts. The assumptions have been thoroughly discussed, refined, and approved by Ansaldo-STS.

2.3 The VELOS Specification

A specification of LDS consists of (1) an *entity description* of the physical and logical entities of LDS, and (2) a *configuration* describing a particular physical layout (relation between the entities). The entity description is specified in a structured programming language, called the VELOS language, that resembles the C++ language. It contains classes that define component, point, EOA, and RBS. Like typical C++ classes, each of these classes contains member variables and member functions. The values of member variables constitute the states of the corresponding entity, while the member functions are used to modify the states of the entity. In particular, the functions of the RBS class define the logic of RBS. Member functions can contain loops that can statically be unrolled. They are also non-recursive, and thus can be inlined. Operations in these functions only involve data of types Boolean, bounded integers, or enumerations, with no pointers and no dynamic memory allocations. (These restrictions are standard for this kind of applications.) The properties to verify are expressed in a temporal logic, and are attached to the corresponding classes in the entity description.

The configuration specifies instances of the classes in the entity description as well as the relation between these instances. In particular, it describes the RBS's and entities that constitute them. Currently, both the entity descriptions and the configurations are created manually by the design engineers. The actual assembly code deployed on the physical devices is automatically generated from the specifications.

3 Verification Techniques and Tools

The problem of selecting the right techniques and tools for the verification of LDS is non-trivial. First, the techniques and tools must scale to industrial-sized designs. In particular, the chosen techniques should address the state explosion problem that is very common in such applications. Second, the right techniques are often not apparent due to a representational issue. The product development team have their own specification language that is far from the languages assumed by most verification techniques or

tools. Third, the high-efficiency demand from the development team often cannot be met by existing techniques and tools. Customizations of and extensions to the existing techniques and tools, as well as a good strategy in combining them, are required to boost the performance.

In this work we appeal to symbolic (software) model checking techniques. Being completely automatic, model checking techniques can easily be integrated into the development cycle. Moreover, symbolic techniques are known to be effective in combating the state explosion problem. In what follows, we review some state of the art symbolic software model checking techniques: bounded model checking (BMC) [7] and counter-example guided abstraction refinement (CEGAR) [15]. We focus on model checking safety properties in the form of program assertion. Although, they are often used to represent requirements, they can also be used to generate execution traces that help the generation of test cases for automatic test pattern generation [22].

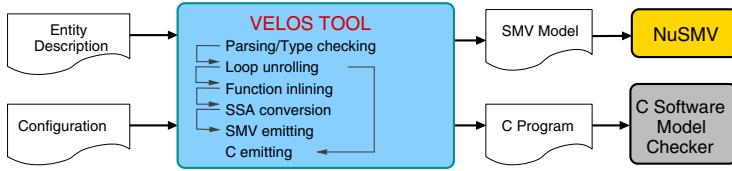
Software model checking has proved to be an effective technique for verifying sequential programs. In particular advances in solvers for satisfiability modulo theory (SMT) [2] have enabled for efficient Boolean reasoning and abstraction computation, which in turn have enabled SMT-based software model checkers to efficiently verify large programs with significant improvements in precision and accuracy.

In the BMC approach one verifies the program by specifying some bounds to guarantee termination. The bounds can be the number of executed statements, the depth of recursions, or the number of loop unwindings. This approach can only be used to disprove assertions (or bug finding). The seminal work on temporal induction [19] uses BMC, not only to disprove properties, but also to prove invariants.

In the CEGAR paradigm one checks if an abstraction (or over-approximation) of the program has an abstract path leading to an assertion violation. If such a path does not exist, then the program is safe. When such a path exists and is feasible in the concrete program, then the path is a counter-example witnessing the assertion violation. Otherwise, the unfeasible path is analyzed to extract information needed to refine the abstraction. For the CEGAR approach, two predicate abstraction based techniques have proved to be effective: the eager abstraction [17] and the lazy abstraction [4]. Predicate abstraction [23] is a technique for extracting a finite-state program from a potentially infinite one by approximating possibly infinite sets of states of the latter by Boolean combinations of some predicates. In each CEGAR iteration of the eager abstraction, one verifies a *Boolean program* extracted from the input program based on a set of predicates. The Boolean program consists only of Boolean variables, each of which corresponds to a predicate used in the abstraction. The lazy predicate abstraction is based on the construction and analysis of an abstract reachability tree (ART). The ART represents an over-approximation of reachable states obtained by unwinding the control-flow graph (CFG) of the program. An ART node typically consists of a location in the CFG, a call stack, and a formula representing a region or a set of data states. The formula in an ART node is obtained by means of predicate abstraction.

4 Verification Approach

We explore two directions for the verification of a VELOS specification: translation into a behaviorally equivalent NUSMV model, and translation into a behaviorally equivalent

**Fig. 3.** Verification and validation flow

C program. The translation into a NUSMV model enables the use of a rich portfolio of verification techniques, e.g., BMC [7], temporal induction [19], CEGAR [15], all available in an extended version of the NUSMV model checker [12]. The translation into a sequential C program enables the use of mature off-the-shelf software model checking techniques and tools. Figure 3 shows our verification and validation flow.

To support our approach, we have implemented a translator, called the VELOS tool, that takes a VELOS specification as an input, and outputs a NUSMV model or sequential C programs. Independently of the final output, the VELOS tool always starts with parsing and type checking. During this phase all the syntactic and semantic errors, if any, are detected and reported to the user. The remaining steps deal with the specific target verification language. In what follows we describe the details of the translation, as well as some issues related to verifying multiple assertions using existing software model checkers. We also explain a customization of and an extension to our C model checker, KRATOS, to efficiently prove multiple assertions simultaneously.

4.1 From VELOS Specifications to NUSMV Models

To verify the VELOS specification, we have to model the application-environment loop. We model such an interaction in the NUSMV model using the synchronous semantics of NUSMV. That is, each transition in the model corresponds to a complete iteration of the loop. Note that this loop may contain an inner loop induced by the scheduler. This latter loop (also called the *scheduler loop*), however, is expected to be finite, and often its termination can be proved by means of simple syntactic criteria. For our case study, we are able to unroll completely the scheduler loop.

The translation of a VELOS specification to a NUSMV model is a complex transformation because of the difference between the paradigm of the two languages. A VELOS specification is in an imperative sequential language with control flow branches and variables assignments. NUSMV language is a transition-based data-flow language, where, for each variable, it is necessary to specify the precise transition relation between the current and next time step variables, including the frame conditions. Moreover, the flow of control in the sequential language has to be encoded explicitly in the NUSMV model using a program-counter variable.

The translation into NUSMV models first removes loops and function calls by, respectively, unrolling and inlining them. Such removals are possible due to the restriction in the VELOS specification, as described in Section 2.3. Finally, the resulting sequential program is translated into its static single assignment (SSA) [18] form, where each variable can be assigned at most once. Such a form reflects the final transition relation

```

if (a > b) a += 2;
else b = a;

```

a_1 = a_0 + 2;
b_1 = a_0;
a_2 = (a_0 > b_0) ? a_1 : a_0;
b_2 = (a_0 > b_0) ? b_0 : b_1;

Fig. 4. Fragment of code (left), and corresponding SSA (right)

in the current and next time steps. For example, for the fragment of code on the left of Figure 4, the SSA conversion would generate the code on the right. In this translation we assumed the initial value of variables a and b are a_0 and b_0 , respectively. After the SSA conversion, the final value of every variable V is always V_i with the highest index i . This corresponds to the “`ASSIGN next(V) := V_i;`” NUSMV statement. In terms of the application, the value of V_i is the value of V after a complete iteration of the application-environment loop. All others V_j only hold values of intermediate expressions, and thus we can use the “`DEFINE V_j := expression;`” construct to avoid the introduction of explicit variables for their representation, which in turn enable NUSMV to inline them with the defined expressions.

The invariant properties in the VELOS specification are output directly as `INVARSPEC` NUSMV statements [11]. The translation also maintains a one-to-one correspondence between variables in the NUSMV model and the variable in the VELOS specification, e.g., for a class instance \mathcal{I} of class C having class member variable V , a NUSMV variable \mathcal{I}_V is declared. Such a correspondence is important for examining counter-examples when some invariants fail to hold, and also for certification purposes.

To obtain NUSMV models that are amenable to be checked by NUSMV, we apply several optimizations in the VELOS tool. In particular, we reduce the number of variables and perform range analysis over possible values for variables. After this analysis, only the Boolean and enumeration types remain, and both are supported by the NUSMV language. For instance, if a variable has an initial value and is never re-assigned, than the variable can be removed completely by converting it into a constant.

4.2 From VELOS Specifications to C Programs

The translation from VELOS specifications into C programs is simpler than the translation into NUSMV models because the language of VELOS and C are both imperative and have a lot in common. In particular, the VELOS tool converts member variables and member functions of class instances into C global variables and functions. Similar to the translation into NUSMV models, the translation into C programs also keeps a one-to-one correspondence between the variables in the resulting C programs and the variables in the VELOS specification.

The C program consists of a top-level main loop that models the application-environment loop. At the beginning of the loop body, the inputs are read and latched. As a common practice in software model checking, we use an extension of C that includes constructs to model non-deterministic data acquisition. The body performs computations according to the logic of the applications. It contains loops that are expected to be finite, and thus in principle can be unrolled. But in general, the unrolling may depend on the input acquired and on the computations performed within the loop. In particular, an inner loop of the top-level main loop corresponds to the scheduler loop.

```

int b1 = 0, b2 = 0;
assert(b1 != 0);
assert(b2 != 0);

```

(a)

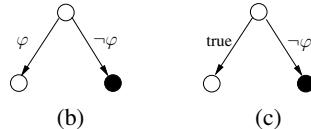


Fig. 5. Fragment of code (a), and assertions as branches in the control-flow graph: standard semantics (b) and assertion-as-property semantics (c)

All properties specified in the VELOS specification are translated into program assertions, and are positioned after all the function calls in the top-level loop body.

4.3 Model Checking Sequential Software with Multiple Assertions

The resulting C programs contain many program assertions. Here, we are interested in checking if *each assertion in the set can be violated*. However, if we put all assertions in a single C program, then it might be the case that some violated assertions can prevent other assertions violation from being detected. Indeed, given an assertion `assert(φ)`, the standard semantics requires that the execution of an assertion will pass if φ holds; otherwise a violation is detected and the execution quits. For instance, in the small program on the left of Figure 5 the second assertion violation can never be detected because the first one is always violated.

We explore two directions in addressing the problem of multiple assertions. First, for each property in the specification, we invoke the VELOS tool to generate a single separate C program such that the only assertion in the program is the property. Second, we invoke the VELOS tool to generate a single C program that contains all properties (or assertions) in the specification, but, during the analysis, we give a different semantics to the assertions such that it allows the execution to escape from assertion violations.

In the first direction, most existing software model checkers for C are readily applicable to verify the generated C programs. However, for a single specification, these model checkers have to be run many times, one run for each of the generated C programs. This bounds to be inefficient. Particularly for the CEGAR-based tools, as the generated C programs come from the same specification, these tools might perform the same abstraction-refinements on these programs.

To overcome these problems, in the second direction, we have extended the sequential analysis of KRATOS to analyze multiple assertions. First, we treat assertions as properties: given an assertion `assert(φ)`, the execution of the assertion can pass whether or not φ holds, but can also quit when φ does not hold. To support this semantics, we customize KRATOS to translate the assertion `assert(φ)` into a different kind of branch in the CFG such that the branch follows the new semantics. That is, instead of translating the assertion into the (b) branch of Figure 5, KRATOS translates the assertion into the (c) branch. With the new translation, checking for an assertion violation can still be reduced to the reachability analysis of the error node (dark node in Figure 5).

Second, we have implemented two different sequential analyses for checking multiple properties simultaneously: *all-in-one-go* and *one-at-a-time*. The all-in-one-go analysis is basically the standard lazy predicate abstraction, but instead of quitting the analysis on finding an assertion violation, the analysis continues the search for other violations. The known violated assertions are no longer considered in the successive

searches. The one-at-a-time analysis, unlike the all-in-one-go, in one run checks one assertion at a time, but uses the ART constructed for checking the previous assertions to prove the subsequent assertions. On finding that an assertion $\text{assert}(\varphi)$ cannot be violated, the corresponding CFG branch is strengthened by turning it into the (b) branch of Figure 5. The one-at-a-time analysis allows for performing an on-the-fly slicing with respect to the checked assertion. Such a slicing can reduce the size of symbolic expressions involved in the abstraction computations, and can also exclude predicates irrelevant to the assertion being checked. The analysis also allows for partitioning the predicates used to prove each assertion, and collecting loop invariants from the constructed ART that will be used to strengthen the successive searches.

5 Experimental Evaluation

5.1 Benchmarks

In our evaluation we consider the 10-RBS case study described in Section 2. To evaluate the scalability of the considered techniques, we create a set of benchmarks by varying the number of RBS's that can be chosen by the scheduler. The RBS's that can be chosen are called *active* RBS's. Given the 10-RBS case study, in total we have 1023 benchmarks with at least one active RBS.

The experiments have been carried out on all of the benchmarks. For presentation, in this paper we report only the statistics obtained from the experiments on two families of 10-RBS benchmarks: one family with one active RBS, subsequently called 1-active-RBS, and the other with all ten active RBS's, subsequently called 10-active-RBS. Experiments on other N -active-RBS families, for $1 < N < 10$, exhibit patterns of statistics that are similar to either that of the 1-active-RBS or that of the 10-active-RBS. The 1-active-RBS family consists of 10 benchmarks and the 10-active-RBS family consists of only 1 benchmark. As explained in Section 2, each benchmark has 53 assertions.

To experiment with NUSMV, the VELOS tool generates an NUSMV model for each of the benchmarks. Each of 1-active-RBS NUSMV models consists of about 7 KLOC and has about 350 elementary Boolean variables. The 10-active-RBS NUSMV model consists of about 60 KLOC and has about 625 elementary Boolean variables.

To evaluate the software model checking approach, as explained in Section 4.3, the VELOS tool has to generate a single C program for each of the assertions. Thus, for the 1-active-RBS family, we have 530 C programs, and, for the 10-active-RBS family, we have 53 C programs. Each of the generated C program is of size 40 KLOC.

5.2 Setup and Configurations

For evaluating the software model checking approach, we experimented with KRATOS, BLAST-2.7 [8], and CPAchecker [5] for the lazy predicate abstraction, with SATABS-3.0 [17] (with CADENCE SMV as the back-end) for the eager predicate abstraction, and with CBMC-4.0 [16] for the BMC approach. For SATABS, BLAST, and CPAchecker, we used the configurations that the tools used in the TACAS 2012 software verification competition. For CBMC, the loop unwinding was limited to the least value sufficient for detecting all assertion violations. For the direction via translation into NUSMV models, we experimented with the BMC algorithm implemented in NUSMV. (We disabled the counter-example generation and we set the bound to five steps.) To prove/disprove

the assertions we ran all the software model checkers but CBMC. For CBMC and NUSMV, we focused on detecting assertion violations using BMC techniques.

We set the time limit to 500s and the memory limit to 2Gb for the experiments with the 1-active-RBS family, and we increased the time limit to one hour and the memory limit to 10Gb for the 10-active-RBS family. All experiments have been performed on a machine equipped with a 2.5GHz Intel Xeon E5420 running Scientific Linux.

5.3 Results of Experiments

The 1-active-RBS Case. The following table shows the results of experiments with the software model checkers on the 1-active-RBS family:

All Properties	KRATOS	BLAST	SATABS	CPACHECKER
Solved	530	0	244	312
Safe	436	0	244	218
Unsafe	94	0	0	94
Time out	0	56	286	2
Memory out	0	474	0	216
Total time	27m:26s	-	3h:58m:30s	1h:48m:30s
Max time	6.7s	-	221.3s	40.3s
Avg. time	3.1s	-	58.6s	20.8s
Max space	147.7Mb	-	454.6Mb	1.3Gb
Avg. space	74.8Mb	-	168.1Mb	985.5Mb

The row “Solved” indicates the number of benchmarks (or the number of assertions) that can be proved/disproved. The rows “Safe” and “Unsafe” indicate the number of, respectively, safe and unsafe benchmarks out of the solved ones. The rows “Time out” and “Memory out” indicate the number of benchmarks on which the tools went out of time and out of memory, respectively. The rows “Total time”, “Max time”, and “Avg. time” indicate, respectively, the total, the maximum, and the average time that the tools used for the solved benchmarks. Similarly for the rows “Max space” and “Avg. space”.

Only KRATOS was able to solve all assertions. BLAST was unable to verify any assertion; it either went out of time/memory, experienced failure in refinement, or crashed. SATABS proved some assertions, but failed to disprove any. As mentioned before, SATABS employs the eager predicate abstraction that requires it to create a Boolean program at each CEGAR iteration. Although SATABS generates such a Boolean program efficiently, given that each benchmark is reasonably big, along with a large number of predicates in some cases, the resulting Boolean program is often too big for the back-end model checker, and thus SATABS spends a lot of time in model checking the generated Boolean programs. Moreover, unlike KRATOS that employs large-block encoding (LBE) [3] on the CFG, SATABS uses basic-block encoding (BBE). As shown in [3], LBE tackles the problem of exploring a huge number of paths in the CFG.

Compared to CPACHECKER, KRATOS showed a better performance on the 1-active-RBS family. Unlike CPACHECKER, KRATOS performs aggressive, but cheap, static program optimizations (e.g., constant propagation, dead-code and unreachable-code eliminations) before analyzing the input program using the lazy predicate abstraction. These optimizations turns out to boost considerably the abstraction computations and the path feasibility analysis. KRATOS can even prove some assertions by simply relying

on these static optimizations and without performing the lazy predicate abstraction at all. CPAchecker employs a flexible LBE on the CFG called adjustable-block encoding (ABE) [6]. The lazy predicate abstraction with ABE often suffers from memory problem because it has to keep track of the results of symbolic evaluations on the CFG paths in the ART nodes.

Focusing on bug hunting, the following table shows the performance of the model checkers in detecting assertion violations:

Unsafe Properties	KRATOS	BLAST	SATABS	CPAchecker	CBMC	NUSMV
Solved	94	0	0	94	94	94
Time out	0	56	94	0	0	0
Memory out	0	38	0	0	0	0
Total time	6m:4s	-	-	42m:6s	22m:58s	59s
Max time	4.9s	-	-	38.9s	18.1s	-
Avg. time	3.9s	-	-	26.9s	14.7s	0.6s
Max space	145.7Mb	-	-	1.2Gb	257.9Mb	30Mb
Avg. space	121.5Mb	-	-	1.1Gb	246.8Mb	25.7Mb

The above table shows that the BMC algorithm in NUSMV performs the best in terms of run time and memory usage. Both CBMC and NUSMV are able to find all assertion violations. However, NUSMV handles enumerative types using a logarithmic encoding (see [12] for details) that turns out to reduce significantly the size of state space.

The 10-active-RBS Case. We now consider the most complex case, i.e. the 10-active-RBS family. The table below contains the accumulated results

All properties	KRATOS	BLAST	SATABS	CPAchecker
Solved	53	0	0	8
Safe	33	0	0	8
Unsafe	20	0	0	0
Time out	0	2	52	0
Memory out	0	43	0	45
Total time	2h:36m:46s	-	-	17m:7s
Max time	553.4s	-	-	208.3s
Avg. time	177.5s	-	-	128.5s
Max space	5.2Gb	-	-	8.4Gb
Avg. space	4.5Gb	-	-	7.9Gb

while the following table focuses only on the case of unsafe properties:

Unsafe properties	KRATOS	BLAST	SATABS	CPAchecker	CBMC	NUSMV
Solved	20	0	0	0	20	20
Time out	0	2	19	0	0	0
Memory out	0	10	0	20	0	0
Total time	26m:45s	-	-	-	2h:41m:22s	129s
Max time	85.7s	-	-	-	8m:26s	-
Avg. time	80.3s	-	-	-	8m:4s	6s
Max space	5.2Gb	-	-	-	728.1Mb	176Mb
Avg. space	5.2Gb	-	-	-	684.3Mb	176Mb

To a large extent, the above results emphasize the pattern discussed for the 1-active-RBS case (and for the intermediate n-active-RBS cases, not reported here for lack of space). Although the state-of-the-art software model checking techniques and tools are feasible for dealing with our case study, they are still far from being efficient in verifying our industrial-sized benchmarks. On the one hand, the CEGAR-based software model checking techniques can readily be used to prove or disprove assertions, but they are far less efficient than the BMC approach via the translation into NUSMV models in bug hunting. On the other hand, the BMC approach is known to be ineffective in proving assertions because one needs to know the diameter of the state space.

We also consider the impact of the new multiple-assertion analyses implemented in KRATOS. The effectiveness of these two new analyses depends heavily on the predicates used or discovered in the abstraction-refinements. In the all-in-one-go analysis, the predicates can simultaneously rule out some assertions violations, but can also clutter the search. The one-at-a-time analysis can benefit from the smaller size of the program resulting from the on-the-fly slicing, and possibly from a small number of predicates resulting from predicate partitioning. However, due to the slicing, the predicates used for solving previous assertions are often not sufficient for discharging the remaining ones. Thus, for solving the remaining assertions, the one-at-a-time analysis needs to further refine the abstraction. Overall, both techniques yield substantial speed-ups of about 80%.

Multiple assertions	KRATOS One-Per-File	KRATOS One-At-A-Time	KRATOS All-In-One-Go
Total time	2h:36m:46s	28m:46s	33m:36s
Max space	5.2Gb	6.3Gb	7.9Gb

Following the above results, we experimented with a simple strategy to further speed up the overall verification process. The idea is to combine several approaches by using cheap techniques to solve as many assertions as possible, and then use expensive ones to prove or disprove the remaining assertions. In particular, we first ran the BMC of NUSMV with a short time limit, to find as many assertion violations as possible, and then used the multiple-assertion analyses of KRATOS to solve the remaining assertions. The results are reported in the following table:

	NUSMV + One-At-A-Time	NUSMV + All-In-One-Go
NUSMV time	129 sec	129 sec
KRATOS time	560 sec	289 sec
Total time	11m:29s	6m:58s
Max space	5.4Gb	5.7Gb

Compared to using multiple-assertion analyses alone, in terms of run time, we further gained a speed-up of up to 80%. We also observe a positive impact on memory usage, with a reduction of up to 28%.

5.4 Remarks on other Experiments

The translation into NUSMV models allows for proving assertions by means of temporal induction. We experimented with the temporal induction implemented in NUSMV to first prove or disprove as many assertions as possible in the 10-active-RBS benchmark, and then use the multiple-assertion analyses to solve the rest. For this experiment,

we only tried with induction of length 10. In this experiment, in addition to detecting all assertion violations as before, NuSMV was able to prove eight assertions. The following table shows that the induction slows down the verification but, with less properties to prove, KRATOS, with the all-in-one-go analysis, consumes considerably less memory than before, i.e., up to 74% of reduction.

	NuSMV + One-At-A-Time	NuSMV + All-In-One-Go
NuSMV time	196 sec	196 sec
KRATOS time	620 sec	254 sec
Total time	13m:36s	7m:30s
Max space	5.4Gb	0.9Gb

Finally, we remark that we also did experiments with explicit-state model checking techniques. Besides model checking, KRATOS is able to encode C programs into PROMELA models that can be checked by the SPIN model checker [25]. However, it turned out that the size of the resulting PROMELA models is beyond the capability of SPIN: SPIN failed to translate the models into pan protocol analyzers.

6 Related Work

There have been numerous works that attempt to apply model checking for the verification of industrial software. A comprehensive survey can be found in [20]. Particularly for the verification of ERTMS, the work closely related to ours includes [1][13][24]. The work in [13] covers the whole safety logic of the interlocking application via manual translations into PROMELA, the language of the SPIN model checker. Besides verifying safety-related properties, the work also verifies liveness properties of the scheduler. Unlike our work, this work creates an under-approximation of the non-deterministic environment, and thus cannot show the absence of bugs.

Similar to [13], the work in [24] considers the whole safety logic, but only verifies bounded safety properties. The work relies on the translation into the target language accepted by the Verus model checker [10]. Both approaches in [13][24] showed poor scalability: they were applied only to small configurations with at most three processes.

The work in [1] shows the use of BMC approaches, via CBMC, to automatically generate test suites for the coverage analysis of safety-critical ERTMS. The problem of generating test suite can be reduced to the problem of verifying multiple assertions. However, due to the bound in loop unwindings, achieving full coverage is hard.

Other works related to the verification of ERTMS include [21][26][27]. The work in [27] applies theorem proving techniques to prove properties in European Train Control System specifications by means of manual encodings into Keymaera. The work in [21] adopts model-based testing, complemented with abstract interpretation techniques, for the verification of a railway signaling system. The work in [26] validates ERTMS specifications via translations into UML, and then uses Petri Nets models to generate test scenario.

Related to the multiple-assertion analysis, the work in [4] describes a technique similar to the all-in-one-go analysis in this paper. However, instead of targeting the verification of multiple properties, the goal of that work is to generate a test suite guaranteeing target predicate coverage.

7 Conclusions and Future Work

We have presented an application of model checking techniques to the verification of a significant fragment of Logica di Sicurezza, the safety logic of an ERTMS Level-2 system developed by Ansaldo-STS. We have developed a verification approach, and performed an evaluation of different verification techniques and tools. From this evaluation, we have learned two main lessons. First, even though existing verification techniques and technologies are readily applicable in the industrial settings, they might be neither efficient nor effective in verifying benchmarks coming from these settings. Moreover, a single approach alone is often not sufficient to handle the benchmarks or to satisfy the high-efficiency demand from the product development team. Second, an appropriate combination of approaches/techniques/technologies can dramatically improve the verification of industrial benchmarks. But such a combination can only be obtained by a thorough experimental evaluation on existing or new techniques.

In this work we have extended the KRATOS software model checker with two analyses that allow for checking multiple assertions simultaneously. We have successfully found a strategy that can handle our case study effectively. That is, we benefit from the efficiency of BMC for ruling out as many assertion violations as possible, and then check the rest of the assertions using the above KRATOS analyses. This strategy has proved to greatly reduce the verification effort.

For future work, given the very promising results, Ansaldo-STS is currently collaborating with FBK to integrate the methodology and approach in its internal Development and V&V Flow in order to verify the whole safety logic, with a possibility to verify the correctness regardless of the configuration.

We also plan to evaluate new model checking techniques, like Property Driven Reachability [9]. Moreover, by exploiting the functionality of NUSMV to dump verification problems in AIGER format, we plan to experiment with other model checkers.

Finally, for the certification of the application, as mandated by the standards, we will work on the certification and qualification of KRATOS and NUSMV.

References

1. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Autom. Reason.* 45(4), 397–414 (2010)
2. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability, Frontiers in Art. Int. and Applications*, vol. 185, pp. 825–885. IOS Press (2009)
3. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. *STTT* 9(5-6), 505–525 (2007)
5. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) *CAV 2011. LNCS*, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
6. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Bloem, R., Sharygina, N. (eds.) *FMCAD*, pp. 189–197. IEEE (2010)

7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
8. Blast-2.7, <http://forge.ispras.ru/projects/blast>
9. 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)
10. Campos, S.V.A., Clarke, E.: The verus language: representing time efficiently with bdds. *Theor. Comput. Sci.* 253(1), 95–118 (2001)
11. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV User Manual v 2.5 (2011), <http://nusmv.fbk.eu>
12. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. *STTT* 2(4), 410–425 (2000)
13. Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F., Traverso, P.: Formal verification of a railway interlocking system using model checking. *Formal Asp. Comput.* 10(4), 361–380 (1998)
14. Cimatti, A., Griggio, A., Micheli, A., Narasamdy, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)
15. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50(5), 752–794 (2003)
16. Clarke, E., Koning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podleski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
17. Clarke, E., Koning, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13(4), 451–490 (1991)
19. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.* 89(4), 543–560 (2003)
20. Fantechi, A., Gnesi, S.: On the Adoption of Model Checking in Safety-Related Software Industry. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 383–396. Springer, Heidelberg (2011)
21. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A., Tempestini, M.: Adoption of model-based testing and abstract interpretation by a railway signalling manufacturer. *IJERTCS* 2(2), 42–61 (2011)
22. Gargantini, A., Heitmeyer, C.L.: Using Model Checking to Generate Tests from Requirements Specifications. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, pp. 146–162. Springer, Heidelberg (1999)
23. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
24. Hartonas-Garmhausen, V., Campos, S.V.A., Cimatti, A., Clarke, E., Giunchiglia, F.: Verification of a safety-critical railway interlocking system with real-time constraints. *Sci. Comput. Program.* 36(1), 53–64 (2000)
25. Holzmann, G.J.: Software model checking with SPIN. *Adv. in Comp.* 65, 78–109 (2005)
26. Jabri, S., El Koursi, E., Bourdeaudhuy, T., Lemaire, E.: European railway traffic management system validation using UML/Petri nets modelling strategy. *European Transp. Res. Review* 2, 113–128 (2010)
27. Platzer, A., Quesel, J.-D.: European Train Control System: A Case Study in Formal Verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)

Minimum Satisfying Assignments for SMT[★]

Isil Dillig¹, Thomas Dillig¹, Kenneth L. McMillan², and Alex Aiken³

¹ College of William & Mary

² Microsoft Research

³ Stanford University

Abstract. A *minimum satisfying assignment* of a formula is a minimum-cost partial assignment of values to the variables in the formula that guarantees the formula is true. Minimum satisfying assignments have applications in software and hardware verification, electronic design automation, and diagnostic and abductive reasoning. While the problem of computing minimum satisfying assignments has been widely studied in propositional logic, there has been no work on computing minimum satisfying assignments for richer theories. We present the first algorithm for computing minimum satisfying assignments for satisfiability modulo theories. Our algorithm can be used to compute minimum satisfying assignments in theories that admit quantifier elimination, such as linear arithmetic over reals and integers, bitvectors, and difference logic. Since these richer theories are commonly used in software verification, we believe our algorithm can be gainfully used in many verification approaches.

1 Introduction

A *minimum satisfying assignment (MSA)* σ of a formula ϕ , relative to a cost function C that maps variables to costs, is a partial variable assignment that entails ϕ while minimizing C . For example, consider the following formula in linear integer arithmetic:

$$\varphi : x + y + w > 0 \vee x + y + z + w < 5 \quad (*)$$

and suppose that the cost function C assigns cost 1 to each variable in the formula. A partial satisfying assignment to this formula is $x = 1, y = 0, w = 0$. This partial assignment has cost 3, since it uses variables x, y, w , each with cost 1. Another satisfying partial assignment to this formula is $z = 0$; this assignment has cost 1 since it only uses variable z . Furthermore, $z = 0$ is a minimum satisfying assignment of the formula since $z = 0 \models \varphi$ and no other satisfying partial assignment assignment of φ has lower cost than the assignment $z = 0$.

Minimum satisfying assignments have important applications in software and hardware verification, electronic design automation, and diagnostic and abductive reasoning [1–3, 6, 4, 5]. For example, in software verification, minimum

* This work was supported by NSF grants CCF-0915766 and CCF-0702681 and DARPA contract 11635025.

satisfying assignments are useful for classifying and diagnosing error reports [6], for finding reduced counterexample traces in bounded model checking [1], and for minimizing the number of predicates required in predicate abstraction [3].

Some applications, such as [1] have used *minimal* rather than *minimum* satisfying assignments (in the context of propositional logic). A minimal satisfying assignment is one from which no variable can be removed while still guaranteeing satisfaction of the formula. Minimal assignments have the advantage that they can be computed greedily by simply removing variables as long as the formula is implied by the remaining assignment. However, a minimal satisfying assignment can be arbitrarily far from optimal, as the following example shows:

Example 1. Consider again the formula φ from (*). The assignment $x = -1, y = -1, w = -1, z = 5$ is *minimal*, with cost 4. That is, removing the assignment to any one variable allows the formula to be false. However, as observed above, the *minimum* cost is 1.

In this paper, we consider the more difficult problem of computing true minimum-cost assignments. This problem has been studied in propositional logic, where it is commonly known as the *minimum prime implicants* problem [4, 5]. However, there has been no work on computing minimum satisfying assignments in the context of satisfiability modulo theories (SMT).

In this paper, we present the first algorithm for computing minimum satisfying assignments for first-order formulas modulo a theory. The algorithm applies to any theory for which full first-order logic, including quantifiers, is decidable. This includes all theories that admit effective quantifier elimination, such as linear arithmetic over reals, linear integer arithmetic, bitvectors, and difference logic. Since these theories and their combinations are commonly used in software verification, we believe an algorithm for computing minimum satisfying assignments in these theories can be gainfully used in many verification approaches.

This paper makes the following key contributions:

- We define minimum satisfying assignments modulo a theory and discuss some useful properties of minimum satisfying assignments.
- We present a branch-and-bound style algorithm for computing minimum satisfying assignments of SMT formulas.
- We consider improvements over the basic branch-and-bound approach that effectively prune parts of the search space.
- We show how to use and compute theory-satisfiable propositional implicants to obtain a good variable order and initial cost bound.
- We describe how to obtain and use a special class of implicants of the original formula to further prune the search space.
- We present an experimental evaluation of the performance of our algorithm.

2 Minimum Satisfying Assignments and Their Properties

To begin with, let us precisely define the notion of MSA for a formula in first-order logic, modulo a theory. This definition is a bit subtle because, to speak of

an assignment of values to variables in first-order logic, we must specify a *model* of the theory.

For a given theory \mathcal{T} , we have a fixed signature \mathcal{S} of predicate and function constants of specified arity. The *theory* \mathcal{T} is a set of first-order sentences over signature \mathcal{S} . A first-order model M is a pair $(\mathcal{U}, \mathcal{I})$ where the set \mathcal{U} is the *universe*, and \mathcal{I} is the *interpretation* that gives a semantics to every symbol in \mathcal{S} . We assume a countable set of *variables* \mathcal{V} , distinct from \mathcal{S} . Given a model M , a *valuation* σ is a partial map from \mathcal{V} to \mathcal{U} . We write $\text{free}(\phi)$ for the set of variables occurring free in formula ϕ . If σ is a valuation in $\text{free}(\phi) \rightarrow \mathcal{U}$, we write $M, \sigma \models \phi$ to indicate that formula ϕ is true, according to the usual semantics of first-order logic, in model M , with σ giving the valuation of the free variables in ϕ . We say M is *model of* theory \mathcal{T} when every sentence of \mathcal{T} is true in M .

Definition 1. (Satisfying assignment) *Formula ϕ is satisfiable modulo \mathcal{T} when there exists a model $M = (\mathcal{U}, \mathcal{I})$ of \mathcal{T} and an assignment $\sigma \in \text{free}(\phi) \rightarrow \mathcal{U}$ such that $M, \sigma \models \phi$. We say the pair (M, σ) is a satisfying assignment for ϕ .*

Our intuition behind an MSA for ϕ is that it gives a valuation for a minimum-cost *subset* of the free variables of ϕ , such that ϕ is true for all valuations of the remaining variables. We capture this idea with *satisfying partial assignments*:

Definition 2. (Satisfying partial assignment) *A satisfying partial assignment for formula ϕ is a pair (M, σ) , where $M = (\mathcal{U}, \mathcal{I})$ is a model, σ is a valuation over M such that $\text{dom}(\sigma) \subseteq \text{free}(\phi)$, and such that for every valuation $\rho \in (\text{free}(\phi) \setminus \text{dom}(\sigma)) \rightarrow \mathcal{U}$, $(M, \sigma \cup \rho)$ is a satisfying assignment for ϕ .*

The following is an alternate statement of this definition:

Proposition 1. *A satisfying partial assignment for formula ϕ is a satisfying assignment for the formula $\forall X. \phi$, for some $X \subseteq \text{free}(\phi)$.*

Now, in order to define a *minimum* partial assignment, we introduce a cost function over partial assignments. For a given function $C \in \mathcal{V} \rightarrow \mathbb{N}$, the cost of a set of variables X is $C(X) = \sum_{v \in X} C(v)$ and the cost of a valuation σ is $C(\sigma) = C(\text{dom}(\sigma))$. Minimum satisfying assignments are now defined as follows:

Definition 3. (Minimum satisfying assignment) *Given a cost function $C \in \mathcal{V} \rightarrow \mathbb{N}$, a minimum satisfying assignment (MSA) for formula ϕ is a partial satisfying assignment (M, σ) for ϕ minimizing $C(\sigma)$.*

As observed in Proposition 1, a partial satisfying assignment (M, σ) for ϕ is just a satisfying assignment for $\forall X. \phi$. We observe that the free variables in this formula are $\text{free}(\phi) \setminus X$, therefore $\text{dom}(\sigma) = \text{free}(\phi) \setminus X$. Minimizing the cost of σ is thus equivalent to *maximizing* the cost of X such that $\forall X. \phi$ is satisfiable. We can formalize this idea as follows:

Definition 4. (Maximum universal subset) *A universal set for formula ϕ modulo theory \mathcal{T} is a set of variables X such that $\forall X. \phi$ is satisfiable. For a given cost function $C \in \mathcal{V} \rightarrow \mathbb{N}$, a maximum universal subset (MUS) is a universal set $X \subseteq \text{free}(\phi)$ maximizing $C(X)$.*

MUS's and MSA's are related by the following theorem:

Theorem 1. *An MSA of formula ϕ for a given cost function C is precisely a satisfying assignment of $\forall X. \phi$ for some MUS X .*

Proof. By Proposition 1, a set $X \subseteq \text{free}(\phi)$ is a universal set exactly when there is a partial satisfying assignment (M, σ) for ϕ such $\text{dom}(\sigma) = \text{free}(\phi) \setminus X$, and therefore $C(\sigma) = C(\text{free}(\phi)) - C(X)$. It follows that X is maximum-cost exactly when σ is minimum-cost. \square

The following corollary follows immediately from Theorem 1.

Corollary 1. *Let σ be an MSA for formula ϕ , and let X be an MUS of ϕ for cost function C . Then,*

$$C(\sigma) = \left(\sum_{v \in \text{free}(\phi)} C(v) \right) - C(X)$$

Universal sets have some useful properties, derived from the properties of universal quantifiers that will aid us in maximizing their cost. First, as stated by Proposition 2, universal sets are downward-closed. Second, as stated by Proposition 3, universal sets are closed under implications:

Proposition 2. *Given a universal set X for formula ϕ , every $X' \subseteq X$ is also a universal set of ϕ .*

Proposition 3. *If X is a universal set for ϕ and $\phi \implies \psi$, then X is a universal set for ψ .*

3 A Branch-and-Bound Algorithm for Computing MSAs

To compute minimum satisfying assignments, we first focus on the problem of finding maximum universal subsets. Since we cannot compute MUSs using a greedy approach, we apply a recursive branch-and-bound algorithm for finding maximum universal sets, shown in Figure 1. This algorithm relies on the downward closure property of universal sets (Proposition 2) to bound the search.

The algorithm `find_mus` of Figure 1 takes as input a formula ϕ , a cost function C , a set of candidate variables V , and a lower bound L , and computes a maximum-cost universal set for ϕ that is a subset of V , with cost greater than L . If there is no such subset, it returns the empty set. The lower bound allows us to cut off the search in cases where the best result thus far cannot be improved.

At each recursive call, the algorithm evaluates at line 1 whether the given lower bound can be improved using the available candidate variables. If not, it gives up and returns the empty set. Otherwise, if there are remaining candidates, it chooses a variable x from the candidate set V (line 3) and decides whether the cost of the universal subset containing x is higher than the cost of the universal subset not containing x (lines 4–10).

At lines 4 – 7, the algorithm determines the cost of the universal subset containing x . Before adding x to the universally quantified subset, we test whether

```

Requires:  $\phi$  is satisfiable
find_mus( $\phi$ ,  $C$ ,  $V$ ,  $L$ ) {
1. If  $V = \emptyset$  or  $C(V) \leq L$  return  $\emptyset$  /* cannot improve bound */
2. Set best =  $\emptyset$ 
3. choose  $x \in V$ 

4. if( $\text{SAT}(\forall x.\phi)$ ) {
5.   Set  $Y = \text{find\_mus}(\forall x.\phi, C, V \setminus \{x\}, L - C(x))$ ;
6.   Int cost =  $C(Y) + C(x)$ 
7.   If (cost >  $L$ ) { best =  $Y \cup \{x\}$ ;  $L = \text{cost}$  }
   }
8. Set  $Y = \text{find\_mus}(\phi, C, V \setminus \{x\}, L)$ ;
9. If ( $C(Y) > L$ ) { best =  $Y$  }

10. return best;
}

```

Fig. 1. Algorithm to compute a maximum universal subset (MUS)

the result is still satisfiable. If not, we give up, since adding more universal quantifiers cannot make the formula satisfiable (the downward closure property of Proposition 2). The recursive call at line 5 computes the maximum universal subset of $\forall x.\phi$, adjusting the cost bound and candidate variables as necessary. Finally, we compute the cost of the universal subset involving x , and if it is higher than the previous bound L , we set the new lower bound to **cost**.

Lines 8 – 9 consider the cost of the universal subset not containing x . The recursive call at line 8 computes the maximum universal subset of ϕ , but the current variable x is removed from the candidate variable set. The algorithm compares the costs of the universal subsets with and without x , and returns the subset with the higher cost.

Finally, the algorithm in Figure 2 computes an MSA of ϕ by using **find_mus**. Here, we first test whether ϕ is satisfiable. If so, we compute a maximum universal subset X for ϕ , and return a satisfying assignment for $\forall X. \phi$, as described in Theorem 1. This algorithm potentially needs to explore an exponential number of possible universal subsets. However, in practice, the recursion can often be cut off at a shallow depth, either because a previous solution cannot be improved, or because the formula $\forall x. \phi$ becomes unsatisfiable, allowing us to avoid branching. Of course, the effectiveness of these pruning strategies depend strongly on the choice of the candidate variable at line 4 as well as the initial bound L on the cost of the MUS. In the following sections, we will describe some heuristics for this purpose that dramatically improve the performance of the algorithm in practice.

```

find_msa( $\phi$ ,  $C$ ) {
1. If  $\phi$  is unsatisfiable, return “UNSAT”
2. Set  $X = \text{find\_mus}(\phi, C, \text{free}(\phi), 0)$ 
3. return a satisfying assignment for  $\forall X. \phi$ .
}

```

Fig. 2. Algorithm to compute minimum satisfying partial assignment

4 Variable Order and Initial Cost Estimate

The performance of the algorithm described in Section 3 can be greatly improved by computing a good initial lower bound L on the cost of the MUS, and by choosing a good variable order. A good initial cost bound L should be as close as possible to the actual MUS cost in order to maximize pruning opportunities at line 1 of the algorithm from Figure 1. Furthermore, a good variable order should first choose variables x for which $\forall x.\phi$ is unsatisfiable at line 4 of the `find_mus` algorithm, as this choice avoids branching early on and immediately excludes large parts of the search space.

Thus, to improve the algorithm of Section 3, we need to compute a good initial MUS cost estimate as well as a set of variables for which the test at line 4 is likely to be unsatisfiable. Observe that computing an initial MUS cost estimate is equivalent to computing an MSA cost estimate, since these values are related as stated by Corollary 1. Furthermore, observe that if x is not part of an MSA, $\forall x.\phi$ is guaranteed to be satisfiable and the check at line 4 of the algorithm will never avoid branching. Thus, if we choose variables likely to be part of an MSA first, there is a much greater chance we can avoid branching early on.

Therefore, our goal is to compute a partial satisfying assignment σ that is a reasonable approximation for an MSA of the formula. That is, σ should have cost close to the minimum cost, and the variables that are part of σ should largely overlap with variables part of an MSA. If we can compute such a partial assignment σ in a reasonably cheap way, we can use it to both compute the initial lower bound L on the cost of the MUS, and choose a good variable order by considering variables part of σ first.

4.1 Using Implicants to Approximate MSAs

One very simple heuristic to approximate MSAs is to greedily compute a *minimal* satisfying assignment σ for ϕ , and use σ to approximate both the cost and the variables of an MSA. Unfortunately, as discussed in Section 1, minimal satisfying assignments can be arbitrarily far from an MSA and, in practice, do not yield good cost estimates or good variable orders (see Section 7).

In this section, we show how to exploit Proposition 3 to find partial satisfying assignments that are good approximations of an MSA. Recall from Proposition 3 that, if ϕ' implies ϕ (is an *implicant* of ϕ), then a universal set of ϕ' is also a universal set of ϕ . In other words, if ϕ' is an implicant of ϕ , then a partial satisfying assignment of ϕ' is also a partial satisfying assignment of ϕ . Thus, if we can compute an implicant of ϕ with a low-cost partial satisfying assignment, we can use it to approximate both the cost as well as the variables of an MSA.

The question then is, how can we cheaply find implicants of ϕ with high-cost universal sets (correspondingly, low-cost partial satisfying assignments)? To do this, we adapt methods for computing “minimum prime implicants” of propositional formulas [4, 5], and consider implicants that are conjunctions of literals. We define a \mathcal{T} -*satisfiable implicant* as a conjunction of literals that

propositionally implies ϕ and is itself satisfiable modulo \mathcal{T} . We say a *cube* is a conjunction of literals which does not contain any atom and its negation.

Definition 5. (\mathcal{T} -satisfiable implicant) Let \mathcal{B}_ϕ be a bijective function from each atom in \mathcal{T} -formula ϕ to a fresh propositional variable. We say that a cube π is a \mathcal{T} -satisfiable implicant of ϕ if (i) $\mathcal{B}_\phi(\pi)$ is a propositional implicant of $\mathcal{B}_\phi(\phi)$ and (ii) π is \mathcal{T} -satisfiable.

Of course, for an implicant to be useful for improving our algorithm, it not only needs to be satisfiable modulo theory \mathcal{T} , but also needs to have a low-cost satisfying assignment. It would defeat our purpose, however, to optimize this cost. Instead, we will simply use the cost of the free variables in the implicant as a trivial upper bound on its MSA. Thus, we will search for implicants whose free variables have low cost.

Definition 6. (Minimum \mathcal{T} -satisfiable implicant) Given a cost function $C : \mathcal{V} \rightarrow \mathbb{N}$, a minimum \mathcal{T} -satisfiable implicant of formula ϕ is a \mathcal{T} -satisfiable implicant π of ϕ minimizing $C(\text{free}(\pi))$.

Example 2. Consider the formula

$$(a + b \geq 0 \vee 2c + d \leq 10) \wedge (a - b \leq 5)$$

For this formula, $a + b \geq 0 \wedge a - b \leq 5$ and $2c + d \leq 10 \wedge a - b \leq 5$ are both \mathcal{T} -satisfiable implicants. However, only $a + b \geq 0 \wedge a - b \leq 5$ is a minimum \mathcal{T} -satisfiable implicant (with cost 2).

To improve the algorithm from Section 3, what we would like to do is to compute a minimum \mathcal{T} -satisfiable implicant for formula ϕ , and use the cost and variables in this implicant as an approximation for those of an MSA of ϕ . Unfortunately, the problem of finding true minimum \mathcal{T} -satisfiable implicants subsumes the problem of finding minimum propositional prime implicants, which is already Σ_2^p -complete. For this reason, we will consider a subclass of \mathcal{T} -satisfiable implicants, called *monotone implicants*, whose variable cost can be optimized using SMT techniques.

To define monotone implicants, we consider only quantifier-free formulas in negation-normal form (NNF) meaning negation is applied only to atoms. If ϕ is not originally in this form, we assume that quantifier elimination is applied and the result is converted to NNF.

Definition 7. (Minimum \mathcal{T} -satisfiable monotone implicant) Given a bijective map \mathcal{B}_ϕ from literals of ϕ to fresh propositional variables, let ϕ^+ denote ϕ with every literal l in ϕ replaced by $\mathcal{B}_\phi(l)$. We say a cube π is a monotone implicant of ϕ if π^+ implies ϕ^+ . A minimum \mathcal{T} -satisfiable monotone implicant of ϕ is a monotone implicant that is \mathcal{T} -satisfiable and minimizes $C(\text{free}(\pi))$ with respect to a cost function C .

To see how monotone implicants differ from implicants, consider the formula $\phi = p \vee \neg p$. Clearly TRUE is an implicant of ϕ . However, it is not a monotone

implicant. That is, suppose that \mathcal{B}_ϕ maps literals p and $\neg p$ to fresh propositional variables q and r respectively, thus $\phi^+ = q \vee r$. This formula is *not* implied by TRUE. In fact, the only monotone implicants are p and $\neg p$. In general, every monotone implicant is an implicant, but not conversely.

4.2 Computing Minimum \mathcal{T} -satisfiable Monotone Implicants

Our goal is to use minimum \mathcal{T} -satisfiable monotone implicants to compute a conservative upper bound on MSA cost and guide variable selection order. In this section, we describe a practical technique for computing minimum \mathcal{T} -satisfiable monotone implicants. Our algorithm is inspired by the technique of [4] and formulates this problem as an optimization problem.

The first step in our algorithm is to construct a boolean abstraction ϕ^+ of ϕ as described in Definition 7. Observe that this boolean abstraction is different from the standard boolean skeleton of ϕ in that two atoms A and $\neg A$ are replaced with different boolean variables. We note that a satisfying assignment for ϕ^+ corresponds to a monotone implicant of ϕ , provided it is consistent, meaning that it does not assign true to both A and $\neg A$ for some atom A .

After we construct the boolean abstraction ϕ^+ , we add additional constraints to ensure that any propositional assignment to ϕ^+ is \mathcal{T} -satisfiable. Let \mathcal{L} be the set of literals occurring in ϕ . We add a constraint Ψ encoding theory-consistency of the implicant as follows:

$$\Psi = \bigwedge_{l \in \mathcal{L}} (\mathcal{B}_\phi(l) \Rightarrow l)$$

Note that in particular, this constraint guarantees that any satisfying assignment is consistent. Moreover, it guarantees that the satisfying assignments modulo \mathcal{T} correspond to precisely the \mathcal{T} -satisfiable monotone implicants.

Finally, we construct a constraint Ω to encode the cost of the monotone implicant. To do this, we first introduce a fresh cost variable c_x for each variable x in the original formula ϕ . Intuitively, c_x will be set to the cost of x if any literal containing x is assigned to true, and to 0 otherwise. We construct Ω as follows:

$$\Omega = \bigwedge_{l \in \mathcal{L}} \left(\mathcal{B}_\phi(l) \Rightarrow \left(\bigwedge_{x \in \text{free}(l)} c_x = C(x) \right) \right) \wedge \bigwedge_{x \in \text{free}(\phi)} (c_x \geq 0)$$

The first conjunct of this formula states that if the boolean variable representing literal l is assigned to true, then the cost variable c_x for each variable in l is assigned to the actual cost of x . The second conjunct states that all cost variables must have a non-negative value.

Finally, to compute a minimum \mathcal{T} -satisfiable monotone implicant, we solve the following optimization problem:

$$\text{Minimize: } \sum_k c_k \quad \text{subject to } (\phi^+ \wedge \Psi \wedge \Omega)$$

This optimization problem can be solved, for example, using the binary search technique of [7], and the minimum value of the cost function yields the cost

of the minimum \mathcal{T} -satisfiable monotone implicant. Similarly, the minimum \mathcal{T} -satisfiable monotone implicant can be obtained from an assignment to $(\phi^+ \wedge \Psi \wedge \Omega)$ minimizing the value of the cost function.

5 Using Implicates to Identify Non-universal Sets

Another useful optimization to the algorithm of Section 3 can be obtained by applying the contrapositive of Proposition 3. That is, suppose that we can find a formula ψ that is implied by ϕ (that is, an *implicate* of ϕ). If Y is not a universal set for ψ then it cannot be a universal set for ϕ . This fact can allow us to avoid the satisfiability test in line 4 of Algorithm `find_mus`, since as soon as our proposed universal subset X contains Y , we know that $\forall X. \phi$ must be unsatisfiable. To use this idea, we need a cheap way to find implicates of ϕ that have small *non-universal* sets.

To make this problem easier, we will consider only theories \mathcal{T} that are *complete*. This means that all the models of \mathcal{T} are *elementarily equivalent*, that is, they satisfy the same set of first-order sentences. Another way to say this is that \mathcal{T} entails every sentence or its negation. An example of a complete theory is Presburger arithmetic, with signature $\{0, 1, +, \leq\}$. Given completeness, we have the following proposition:

Proposition 4. *Given a complete theory \mathcal{T} , and formula ψ , if $\neg\psi$ is satisfiable modulo \mathcal{T} , then $\forall \text{free}(\psi). \psi$ is unsatisfiable modulo \mathcal{T} .*

Proof. Let V be the set of free variables in ψ . Since $\neg\psi$ is satisfiable modulo \mathcal{T} , there is a model M of \mathcal{T} such that $M \models \exists V. \neg\psi$. Since \mathcal{T} is complete, it follows that $\exists V. \neg\psi$ is true in all models of \mathcal{T} , hence $\forall V. \phi$ is unsatisfiable modulo \mathcal{T} . \square

This means that if we can find an implicate ψ of ϕ , such that $\neg\psi$ is satisfiable, then we can rule out any candidate universal subset for ϕ that contains the free variables of ψ . To find such non-trivial implicates, we will search for formulas of the form $\psi = \psi_1 \Rightarrow \psi_2$, where ψ_1 and ψ_2 are built from sub-formulas of ϕ . The advantage of considering this special class of implicates is that they can be easily derived from the boolean structure of the formula.

Specifically, to derive these implicates, we first convert ϕ to NNF and compute a so-called *trigger* Π for each subformula of ϕ . Triggers of each subformula are defined recursively as follows:

1. For the top-level formula ϕ , $\Pi(\phi) = \text{true}$.
2. For a subformula $\phi' = \phi_1 \wedge \phi_2$, $\Pi(\phi_1) = \Pi(\phi')$, and $\Pi(\phi_2) = \Pi(\phi')$.
3. For a subformula $\phi' = \phi_1 \vee \phi_2$, $\Pi(\phi_1) = \Pi(\phi') \wedge \neg\phi_2$, and
 $\Pi(\phi_2) = \Pi(\phi') \wedge \neg\phi_1$.

Example 3. Consider the formula $x \neq 0 \vee (x + y < 5 \wedge z = 3)$. Here, the triggers for each literal are as follows:

$$\begin{aligned}\Pi(x + y < 5) &= \neg(x \neq 0) \\ \Pi(z = 3) &= \neg(x \neq 0) \\ \Pi(x \neq 0) &= \neg(x + y < 5 \wedge z = 3)\end{aligned}$$

It is easy to see that if l is a literal in formula ϕ with trigger $\Pi(l)$, then ϕ implies $\Pi(l) \Rightarrow l$. Thus, $\Pi(l) \Rightarrow l$ is always a valid implicate of ϕ . However, it is not the necessarily the case that $\neg(\Pi(l) \Rightarrow l)$ is satisfiable. To make sure we only obtain implicants where $\neg(\Pi(l) \Rightarrow l)$ is satisfiable, we first convert ϕ to a simplified form defined in [8]. This representation guarantees that for any trigger $\Pi(l)$ of l , $\neg(\Pi(l) \Rightarrow l)$ is satisfiable. Thus, once a formula ϕ has been converted to simplified form, implicants with satisfiable negations can be read off directly from the boolean structure of the formula without requiring satisfiability checks.

If $\psi = \Pi(l) \Rightarrow l$ is an implicate obtained as described above, we know that no universal subset for ϕ contains $\text{free}(\psi)$. Thus, when the last variable in $\text{free}(\psi)$ is universally quantified, we can backtrack without checking satisfiability.

6 Implementation

We have implemented the techniques described in this paper in our Mistral SMT solver available at www.cs.wm.edu/~tdillig/mistral.tar.gz. Mistral solves constraints in the combined theories of linear integer arithmetic, theory of equality with uninterpreted functions, and propositional logic. Mistral solves linear inequalities over integers using the Cuts-from-Proofs algorithm described in [9], and uses the MiniSAT solver as its SAT solving engine [10].

While the algorithm described in this paper applies to all theories that admit quantifier elimination, our implementation focuses on computing minimum satisfying assignments in Presburger arithmetic (linear arithmetic over integers). To decide satisfiability of quantified formulas in linear integer arithmetic, we use Cooper’s technique for quantifier elimination [11]. However, since we expect a significant portion of the universally quantified formulas constructed by the algorithm to be unsatisfiable, we perform a simple optimization designed to detect unsatisfiable formulas: In particular, before we apply Cooper’s method, we first instantiate universally quantified variables with a few concrete values. If any of these instantiated formulas are unsatisfiable, we know that the universally quantified formula must be unsatisfiable.

The algorithm presented in this paper performs satisfiability checks on many similar formulas. Since many of these formulas are comprised of the same set of atoms, the SMT solver typically relearns the same theory conflict clauses many times. Thus, to take advantage of the similarity of satisfiability queries, we reuse theory conflict clauses across different satisfiability checks whenever possible.

For computing minimum \mathcal{T} -satisfiable implicants, we have implemented the technique described in Section 4 and used the binary search technique described in [7] for optimizing the cost function. However, since finding the actual minimum monotone implicant can be expensive (see Section 7.1), our implementation allows terminating the search for an optimal value after a fixed number of steps. In practice, this results in implicants that are not in fact minimum, but “close enough” to the minimum. This approach is sound because the underlying optimization procedure hill climbs from an initial solution towards an optimal one, and the solution at any step of the optimization procedure can be used as a bound on the cost of a minimum \mathcal{T} -satisfiable implicant.

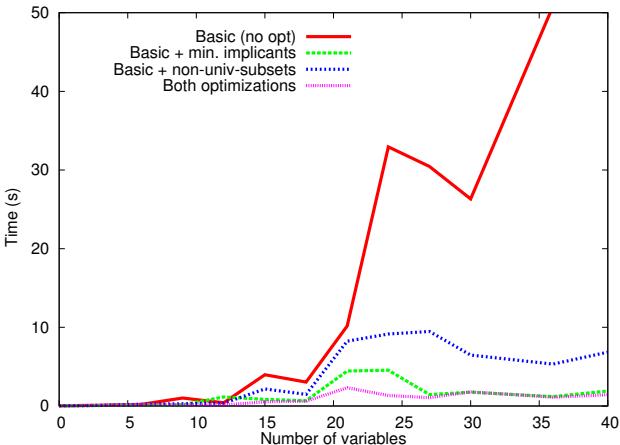


Fig. 3. (# variables in the original formula vs. time to compute MSA in seconds

7 Experimental Results

To evaluate the performance of the algorithm proposed in this paper, we computed minimum satisfying assignments for approximately 400 constraints generated by the program analysis tool Compass [6, 12]. In this application, minimum satisfying assignments are used to compute small, relevant queries that help users diagnose error reports as real bugs or false alarms [6]. In this setting, the number of variables in the satisfying partial assignment greatly affects the quality of queries presented to users. As a result, the time programmers take to diagnose potential errors depends greatly on the number of variables used in the satisfying assignment; thus, computing true minimum-cost assignments is crucial in this setting. The benchmarks we used for our evaluation are available from www.cs.wm.edu/~tdillig/msa-benchmarks.tar.gz.

We chose to evaluate the proposed algorithm on the constraints generated by Compass rather than the standard SMTLIB benchmarks for two reasons: First, unlike the constraints we used, SMTLIB benchmarks are not taken from applications that require computing minimum satisfying assignments. Second, the large majority of benchmarks in the QF_LIA category of the SMTLIB benchmarks contain uninteresting MSAs (containing all or almost all variables in the original formula), making them inappropriate for evaluating an algorithm for computing MSAs.

The constraints we used in our experimental evaluation range in size from a few to several hundred boolean connectives, with up to approximately 40 variables. In our evaluation, we measured the performance of all four versions of the algorithm. The first version, indicated with red in Figures 3 and 4, corresponds to the basic branch-and-bound algorithm described in Section 3. The second version, indicated with green on the graphs, uses the minimum implicant optimization of Section 4. However, as mentioned earlier, since computing

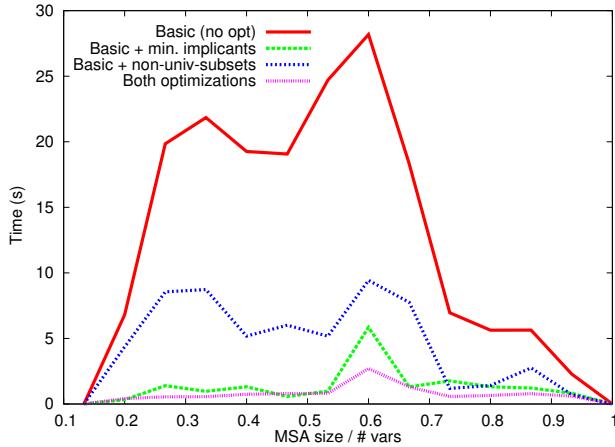


Fig. 4. (# variables in MSA/# of variables in formula) vs. time in seconds

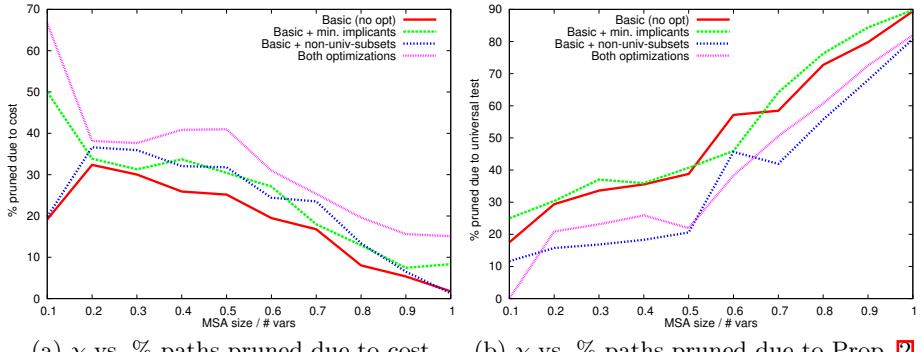
the true minimum implicant can be expensive (see Section 7.1), we only use an approximate solution to the resulting optimization problem. The third version of the algorithm is indicated with blue lines and corresponds to the basic algorithm from Section 3 augmented with the technique of Section 5 for identifying non-universal sets. Finally, the last version of the algorithm using both of the optimizations of Sections 4 and 5 is indicated in the graphs with pink lines.

Figure 3 plots the number of variables in the original formula against running time in seconds for all four versions of the algorithm. As this figure shows, the performance of the basic algorithm is highly sensitive to the number of variables in the original formula and does not seem to be practical for formulas containing more than ~ 18 variables. Fortunately, the improvements described in Sections 4 and 5 have a dramatic positive impact on performance. As is evident from a comparison of the blue, green, and pink lines, the two optimizations of Section 4 and Section 5 complement each other, and we obtain the most performant version of the algorithm by combining both of these optimizations. In fact, the cost of the algorithm using both optimizations seems to grow slowly in the number of variables, indicating that the algorithm should perform well in many settings. However, even using both optimizations, computing MSAs is still much more computationally expensive than deciding satisfiability. On average, computing MSAs is about 25 times as expensive as computing satisfiability on our benchmarks.

Figure 4 plots the fraction

$$\chi = \frac{\text{\#of variables in MSA}}{\text{\#of variables in formula}}$$

against running time in seconds. As this figure shows, if χ is very small (i.e., the MSA is small compared to the number of variables in the formula), the problem of computing minimum satisfying assignments is easy, particularly for

**Fig. 5.** Effectiveness of pruning strategies

the versions of the algorithm using the minimum implicant optimization. Dually, as χ gets close to 1 (i.e., MSA contains almost all variables in the formula, thus few variables can be universally quantified), the problem of computing minimum satisfying assignments again becomes easier. As is evident from the shape of all four graphs in Figure 4, the problem seems to be the hardest for those constraints where χ is approximately 0.6. Furthermore, observe that for constraints with $\chi < 0.6$, the minimum implicant optimization is much more important than the optimization of Section 5. In contrast, the non-universal sets optimization seems to become more important as χ exceeds the value 0.7. Finally, observe that the fully optimized version of the algorithm often performs at least an order of magnitude better than the basic algorithm; at $\chi = 0.6$, the optimized algorithm takes an average of 2.7 seconds, while the basic algorithm takes 28.2 seconds.

Figure 5 explores why we observe a bell-shaped curve in Figure 4. Figure 5(a) plots the value χ against the percentage of all search paths pruned because the current best cost estimate cannot be improved. As this figure shows, the smaller χ is, the more paths can be pruned due to the bound and the more important it is to have a good initial estimate. This observation explains why the minimum implicant optimization is especially important for small values of χ .

In contrast, Figure 5(b) plots the value of χ against the percentage of paths pruned due to the formula ϕ becoming unsatisfiable (i.e., due to Proposition 2). This graph shows that, as the value of χ increases, and thus the MUS's become smaller, more paths are pruned in this way. This observation explains why all versions of the algorithm from Figure 4 perform much better as χ increases.

7.1 Other Strategies to Obtain Bound and Variable Order

In earlier sections, we made the following claims:

1. Computing true minimum-cost implicants is too expensive, but we can obtain a very good approximation to the minimum implicant by terminating the optimizer after a small number of steps
2. *Minimal* satisfying assignments are not useful for obtaining a good cost estimate and variable order

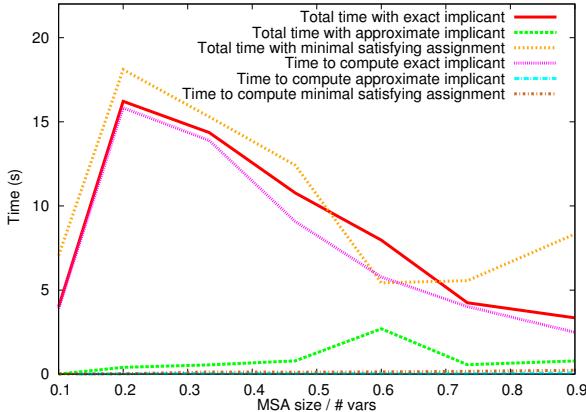


Fig. 6. Comparison of strategies to obtain cost estimate and variable order

In this section, we give empirical data to justify both of these claims.

Figure 6 compares the performance of the algorithm using different strategies to obtain a cost estimate and variable order. As before, the x -axis plots the value of χ and the y -axis is running time in seconds. The red line in this figure shows the total running time of the MSA algorithm using the true minimum-cost monotone implicant. In contrast, the green line shows the total running time of the algorithm using an approximation of the minimum-cost monotone implicant, obtained by terminating the search for the optimum value after a fixed small number of steps. As is evident from this figure, the performance of the algorithm using the true-cost minimum implicant is much worse than the approximately-minimum implicant. This observation is explained by considering the pink line in Figure 6, which plots the time for computing the true minimum-cost monotone implicant. As can be seen by comparing the red and pink lines, the time to compute the true minimum implicant completely dominates the time for the total MSA computation. In contrast, the time to compute the approximate minimum implicant (shown in blue) is negligible, but it is nearly as effective for improving the running time of the MSA algorithm.

We now consider the performance of the algorithm (shown in orange in Figure 6) when we use a *minimal* satisfying assignment to bound the initial cost estimate and choose a variable order. The brown line in the figure shows the time to compute a minimal satisfying assignment. As is clear from Figure 6, the overhead of computing a minimal satisfying assignment is very low, but the performance of the MSA computation algorithm using minimal satisfying assignments is very poor. One explanation for this is that *minimal* satisfying assignments do not seem to be very good approximations for true MSAs. For instance, on average, the cost of a minimal satisfying assignment is 30.6% greater than the cost of an MSA, while the cost of the approximately minimum monotone implicant is only 7.7% greater than the MSA cost. Thus, using minimal satisfying assignments to bound cost and choose a variable order does not seem to be a very good heuristic.

8 Related Work

The problem of computing minimum satisfying assignments in propositional logic is addressed in [4, 13, 5]. All of these approaches formulate the problem of computing implicants as integer linear programming and solve an optimization problem to find a satisfying assignment. Our technique for computing minimum \mathcal{T} -satisfiable monotone implicants as described in Section 4 is similar to these approaches. However, we are not aware of any algorithms for computing minimum satisfying assignments in theories richer than propositional logic.

Minimum satisfying assignments have important applications in program analysis and verification. One application of minimum satisfying assignments is finding concise explanations for potential program errors. For instance, recent work [6] uses minimum satisfying assignments for automating error classification and diagnosis using abductive inference. In this context, minimum satisfying assignments are used for computing small, intuitive queries that are sufficient for validating or discharging potential errors. Similarly, the work described in [1] uses minimal satisfying assignments to make model checking tools more understandable to users. In this context, minimal satisfying assignments are used to derive small, reduced counterexample traces that are easily understandable.

Another important application of minimum satisfying assignments in verification is abstraction refinement. One can think of minimum satisfying assignments in this context as an application of Occam’s razor: the simplest explanation of satisfiability is the best; thus, minimum satisfying assignments can be used as a guide to choose the most relevant refinements. For instance, the work of Amla and McMillan [4] uses an approximation of minimal satisfying assignments, referred to as *justifications*, for abstraction refinement in SAT-based model checking. Similarly, the work presented in [3] uses minimal satisfying assignments for obtaining a small set of predicates used in the abstraction. However, the results presented in [3] indicate that using minimum rather than minimal satisfying assignments might be more beneficial in this context. In fact, the authors themselves remark on the following: “Another major drawback of the greedy approach is its unpredictability . . . Clearly, the order in which this strategy tries to eliminate predicates in each iteration is very critical to its success.”

9 Conclusion

In this paper, we have considered the problem of computing minimum satisfying assignments for SMT formulas, which has important applications in software verification. We have shown that MSAs can be computed with reasonable cost in practice using a branch-and-bound approach, at least for a set of benchmarks obtained from software verification problems. We have shown that the search can be usefully bounded by computing implicants with upper-bounded MSAs and implicants with upper-bounded MUS’s, provided the cost of obtaining these is low. While our optimizations seem effective, we anticipate that significant improvements are possible, both in the basic algorithm and the optimizations.

Expanding the approach to richer theories is also an interesting research direction. The problem of finding MSA modulo \mathcal{T} is decidable when the satisfiability

modulo \mathcal{T} is decidable in the universally quantified fragment of the logic. This is true for a number of useful theories, including Presburger and bitvector arithmetic. While our approach does not apply to theories that include uninterpreted functions, arrays or lists, this problem may be solved or approximated in practice. In this case, it could be that the notion of partial assignment must be refined, so that the cost metric can take into account the complexity of valuations of structured objects such as arrays and lists.

In summary, we believe that the problem of finding MSAs modulo theories will have numerous applications and is a promising avenue for future research.

References

1. Ravi, K., Somenzi, F.: Minimal Assignments for Bounded Model Checking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 31–45. Springer, Heidelberg (2004)
2. Marquis, P.: Extending Abduction from Propositional to First-Order Logic. In: Jorrand, P., Kelemen, J. (eds.) FAIR 1991. LNCS, vol. 535, pp. 141–155. Springer, Heidelberg (1991)
3. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate Abstraction with Minimum Predicates. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 19–34. Springer, Heidelberg (2003)
4. Silva, J.: On computing minimum size prime implicants. In: International Workshop on Logic Synthesis, Citeseer (1997)
5. Pizzuti, C.: Computing prime implicants by integer programming. In: IEEE International Conference on Tools with Artificial Intelligence, pp. 332–336. IEEE (1996)
6. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI (2012)
7. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability Modulo the Theory of Costs: Foundations and Applications. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 99–113. Springer, Heidelberg (2010)
8. Dillig, I., Dillig, T., Aiken, A.: Small Formulas for Large Programs: On-Line Constraint Simplification in Scalable Static Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 236–252. Springer, Heidelberg (2010)
9. Dillig, I., Dillig, T., Aiken, A.: Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 233–247. Springer, Heidelberg (2009)
10. Sörensson, N., Een, N.: Minisat v1. 13-a sat solver with conflict-clause minimization. In: SAT 2005, p. 53 (2005)
11. Cooper, D.: Theorem proving in arithmetic without multiplication. Machine Intelligence 7(91-99), 300 (1972)
12. Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. POPL 46(1), 187–200 (2011)
13. Manquinho, V., Flores, P., Silva, J., Oliveira, A.: Prime implicant computation using satisfiability algorithms. In: ICTAI, pp. 232–239 (1997)
14. Amla, N., McMillan, K.L.: Combining Abstraction Refinement and SAT-Based Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 405–419. Springer, Heidelberg (2007)

When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way

Cheng-Shen Han and Jie-Hong Roland Jiang

Department of Electrical Engineering / Graduate Institute of Electronics Engineering
National Taiwan University, Taipei 10617, Taiwan
hankf4@gmail.com, jhjiang@cc.ee.ntu.edu.tw

Abstract. Recent research on Boolean satisfiability (SAT) reveals modern solvers' inability to handle formulae in the abundance of parity (XOR) constraints. Although XOR-handling in SAT solving has attracted much attention, challenges remain to completely deduce XOR-inferred implications and conflicts, to effectively reduce expensive overhead, and to directly generate compact interpolants. This paper integrates SAT solving tightly with Gaussian elimination in the style of Dantzig's simplex method. It yields a powerful tool overcoming these challenges. Experiments show promising performance improvements and efficient derivation of compact interpolants, which are otherwise unobtainable.

1 Introduction

For over a decade of intensive research, Boolean satisfiability (SAT) solving [2] on conjunctive normal form (CNF) formulae has become a mature technology enabling pervasive applications in hardware/software verification, electronic design automation, artificial intelligence, and other fields. The maturity on the other hand sharpens the boundary between what can and what cannot be achieved by the state-of-the-art solving techniques [23][22][10]. One clear limitation is their poor scalability in solving formulae that in part encode parity (XOR) constraints, which arise naturally in real-world applications such as cryptanalysis [21], model counting [11], decoder synthesis [14], arithmetic circuit verification, etc.

To overcome this limitation, there are prior attempts integrating special XOR handling into SAT solving [27][4][15][6][7][25][16][26][17]. Two different strategies have been explored. Non-interactive XOR handling, on the one hand, as pursued in [27][6][7] performs XOR reasoning and SAT solving in separate phases. Interactive XOR handling, on the other hand, as pursued in [4][15][25][16][26][17] invokes XOR reasoning on-the-fly during SAT solving. Despite the expensiveness of XOR handling compared to CNF handling, positive results on conquering traditionally difficult problems have been demonstrated especially by the latter strategy, which is taken in this paper. Prior interactive methods can be further classified into two categories: inference-rule based [4][15][16][17] and linear-algebra based [25][26] XOR reasoning. The latter tends to be simpler in realization, and can be faster in performance as suggested by the empirical results in [17]. This paper adopts linear-algebra based computation [25][26].

Regardless of the recent progress in XOR-reasoning, several challenges remain to be further addressed. Firstly, the deductive power of XOR-reasoning should be enhanced. To the best of the authors' knowledge, no current solver guarantees complete propagation/conflict detection for a given set of XOR-constraints with respect to some variable assignment. Secondly, the overhead of XOR-reasoning should be reduced, and the synergy between CNF solving and XOR-reasoning should be further strengthened. Thirdly, Craig interpolant generation is not supported by any current solver equipped with the XOR-reasoning capability. As interpolation becomes an indispensable tool for verification [19] and synthesis [13], compact interpolant derivation from combined CNF and XOR reasoning should be solicited.

The efforts of combining CNF and XOR reasoning share a common connection to Satisfiability Modulo Theories (SMT) [24]. There is, however, a subtle difference that makes these efforts distinct. The underlying CNF and XOR handlers encounter the same variables, whereas most, if not all, current SMT solvers with capability of producing Craig interpolants [8] assume the considered theories are of disjoint signatures. This difference makes recent advances in SMT solving and interpolation [20, 28, 5] not immediately helpful to alleviate the aforementioned challenges.

This paper tackles the above three challenges with the following results. Gauss-Jordan elimination (GJE) (in contrast to prior Gaussian elimination (GE) [25, 26]) is proposed for XOR-constraint processing in a matrix form. It admits complete detection of XOR-inferred propagations and conflicts. As the matrix is in the reduced row echelon form, the two-literal watching scheme fits in naturally for fast propagation/conflict detection, and for lazy and incremental matrix update in the style of Dantzig's simplex algorithm [9]. This simple data structure effectively reduces computation overhead and tightens the integration between CNF and XOR reasoning. Moreover, interpolant derivation rules are obtained for direct and compact interpolant generation. Experimental results suggest strong benefit of the proposed method in accelerating SAT solving. Promising improvements over the prior state-of-the-art solver [26] are observed. Moreover the results show efficient derivation of compact interpolants, which are otherwise unobtainable.

This paper is organized as follows. Preliminaries are given in Section 2. Section 3 presents our framework on SAT solving and XOR-reasoning; Section 4 covers interpolant generation in our framework. Experimental results and discussions are given in Section 5. Detailed comparison with the closest related work is performed in Section 6. Finally, Section 7 concludes this paper and outlines future work.

2 Preliminaries

We define terminology and notation to be used throughout this paper. Symbols \wedge , \vee , \neg , and \oplus stand for Boolean AND, OR, NOT, and exclusive OR (XOR) operations, respectively. A *literal* is either a variable (i.e., in the positive phase) or

the negation of a variable (i.e., in the negative phase). A (regular) *clause* is a disjunction of a set of literals. A Boolean formula is in *conjunctive normal form* (CNF) if it is expressed as a conjunction of a set of clauses. For a literal l , its corresponding variable is denoted as $\text{var}(l)$. Also since a clause is viewed as a set of literals, expression $l \in C$ denotes that l is a constituent literal of clause C , and $C' \subseteq C$ denotes C' is a subclause of C .

2.1 XOR Constraints

An XOR-*clause* is a series of XOR operations over a set of literals and/or Boolean constants $\{0, 1\}$. It equivalently represents a linear equation over $\text{GF}(2)$, the Galois field of two elements. An XOR-clause is in the *standard form* if all of its literals appear in the positive phase. E.g., the XOR-clause $(x_1 \oplus \neg x_2 \oplus x_3)$ can be written in the standard form as $(x_1 \oplus x_2 \oplus x_3 \oplus 1)$, which equivalently represents the linear equation $x_1 \oplus x_2 \oplus x_3 = 0$. Note that an XOR-clause consisting of n variables translates into a conjunction of 2^{n-1} clauses with n literals each. E.g., the XOR-clause $(x_1 \oplus \neg x_2 \oplus x_3)$ can be classified to the equivalent CNF $(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. To avoid such exponential translation, an n -element XOR-clause $(l_1 \oplus \dots \oplus l_n)$ can be divided into two XOR-clauses $(l_1 \oplus \dots \oplus l_k \oplus y)$ and $(\neg y \oplus l_{k+1} \oplus \dots \oplus l_n)$ by introducing a new fresh variable y . Some modern SAT solvers, e.g., CRYPTOMINISAT [26], can extract XOR-clauses from a set of regular clauses¹.

A set of m XOR-clauses over n variables $\mathbf{x} = \{x_1, \dots, x_n\}$ can be considered as a system of m linear equations over n unknowns. Hence the XOR-constraints can be represented in a matrix form as $A\mathbf{x} = \mathbf{b}$, where A is an $m \times n$ matrix and \mathbf{b} is an $m \times 1$ constant vector of values in $\{0, 1\}$. In the sequel, $A\mathbf{x} = \mathbf{b}$ is alternatively represented as a single Boolean matrix $M = [A|\mathbf{b}]$, where separation symbol “|” denotes matrix concatenation of A and \mathbf{b} , that is, matrix A is augmented with one more column \mathbf{b} .

Example 1. The three XOR-clauses $c_1: (x_1 \oplus \neg x_4)$, $c_2: (x_2 \oplus x_4)$, and $c_3: (x_1 \oplus \neg x_2 \oplus \neg x_3)$ correspond to the linear equations with the following matrix form.

$$\begin{array}{ccccc} & x_1 & x_2 & x_3 & x_4 & \mathbf{b} \\ c_1 & \left(\begin{array}{ccccc} 1 & 0 & 0 & 1 & 0 \end{array} \right) \\ c_2 & \left(\begin{array}{ccccc} 0 & 1 & 0 & 1 & 1 \end{array} \right) \\ c_3 & \left(\begin{array}{ccccc} 1 & 1 & 1 & 0 & 1 \end{array} \right) \end{array}$$

A matrix M can be reduced by Gaussian or Gauss-Jordan elimination to remove linearly dependent equations. Without loss of generality, we shall assume that matrix M has been preprocessed and is of full rank. In our treatment a matrix is often underdetermined, namely, there are more columns (unknowns) than rows (constraints). In the sequel, a matrix is also viewed as a set of rows.

This paper is concerned with the Boolean satisfiability of a formula given as a conjunction of regular clauses and/or XOR-clauses. Thus a formula is viewed

¹ Our implementation adopts the XOR extraction computation of CRYPTOMINISAT.

as a set of (XOR-)clauses. (In practice, the XOR-clauses can be given as part of the formula or deduced from the regular clauses.) In the sequel, a formula ϕ (respectively a system of linear equations $[A|\mathbf{b}]$) over variables \mathbf{x} subject to some truth assignment $\alpha : \mathbf{x}' \rightarrow \{0, 1\}$ on variables $\mathbf{x}' \subseteq \mathbf{x}$ is denoted as $\phi|_\alpha$ (respectively $[A|\mathbf{b}]|_\alpha$). That is, $\phi|_\alpha$ (respectively $[A|\mathbf{b}]|_\alpha$) is the induced formula of ϕ (respectively linear equations $[A|\mathbf{b}]$) with variable x_i substituted with its truth value $\alpha(x_i)$. We represent α with a characteristic function. E.g., $\alpha = \neg x_1 x_2 \neg x_3$ denotes $\alpha(x_1) = 0$, $\alpha(x_2) = 1$, and $\alpha(x_3) = 0$.

2.2 Resolution Refutation and Craig Interpolation

Assume literal l is in clause C_1 and $\neg l$ in C_2 . A *resolution* of clauses C_1 and C_2 on $\text{var}(l)$ yields a new clause C containing all literals in C_1 and C_2 except for l and $\neg l$. The clause C is called the *resolvent* of C_1 and C_2 . For an unsatisfiable CNF formula, there always exists a resolution sequence, referred to as a *resolution refutation*, leading to an empty-clause resolvent. Resolution refutation has a tight connection to Craig interpolants.

Theorem 1 (Craig Interpolation Theorem). [8]

For two Boolean formulae ϕ_A and ϕ_B with $\phi_A \wedge \phi_B$ unsatisfiable, there exists a Boolean formula I_A referring only to the common variables of ϕ_A and ϕ_B such that $\phi_A \rightarrow I_A$ and $I_A \wedge \phi_B$ is unsatisfiable.

The Boolean formula I_A is referred to as the *interpolant* of ϕ_A with respect to ϕ_B . When ϕ_A and ϕ_B are in CNF, a refutation proof of $\phi_A \wedge \phi_B$ is derivable from a SAT solver such as MINISAT [10]. Further, an interpolant circuit I_A can be constructed from the refutation proof in linear time [20].

3 Satisfiability Solving under XOR Constraints

Modern SAT solvers are based on the conflict-driven clause learning (CDCL) mechanism. Our proposed decision procedure is built on top of the modern solvers. Figure 1 sketches the pseudo code, where lines 2 and 13-16 are inserted for special XOR-handling. In line 2, XOR-clauses are extracted from the input formula ϕ . Let $A\mathbf{x} = \mathbf{b}$ be a system of linearly independent equations derived from these XOR-clauses. Then $M = [A|\mathbf{b}]$. If M is empty, lines 13-16 take no effect and the pseudo code works the same as the standard CDCL procedure. On the other hand, when M contains a non-empty set of linear equations, the procedure *Xorplex* in line 13 deduces implications or conflicts whenever they exist from M with respect to a given variable assignment α . In the process, matrix M may be changed along the computation. When implication (or propagation) happens, α is expanded to include newly implied variables. If any implication or conflict results from *Xorplex*, in line 15 essential information is added to ϕ in the form of learnt clauses, which not only reduces search space but also facilitates future conflict analysis.

SatSolve

```

input: Boolean formula  $\phi$ 
output: SAT or UNSAT
begin
01    $\alpha := \emptyset;$ 
02    $M := ObtainXorMatrix(\phi);$ 
03   repeat
04     ( $\text{status}, \alpha$ ) := PropagateUnitImplication( $\phi, \alpha$ );
05     if  $\text{status} = \text{conflict}$ 
06       if conflict at top decision level
07         return UNSAT;
08        $\phi := AnalyzeConflict \& AddLearnedClause(\phi, \alpha);$ 
09        $\alpha := Backtrack(\phi, \alpha);$ 
10     else
11       if all variables assigned
12         return SAT;
13       ( $\text{status}, \alpha$ ) := Xorplex( $M, \alpha$ );
14       if  $\text{status} = \text{propagation or conflict}$ 
15          $\phi := AddXorImplicationConflictClause(\phi, M, \alpha);$ 
16         continue;
17        $\alpha := Decide(\phi, \alpha);$ 
end

```

Fig. 1. Algorithm: SatSolve

3.1 XOR Reasoning

Before elaborating our XOR-reasoning technique, we show an example motivating the adoption of Gauss-Jordan elimination.

Example 2. Consider the following matrix triangularized by Gaussian elimination.

$$[A|\mathbf{b}] = \left(\begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{array} \right)$$

No implication can be deduced from it. With Gauss-Jordan elimination, however, it is reduced to the following diagonal matrix.

$$[A'|\mathbf{b}'] = \left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{array} \right)$$

The values of the first three variables can be determined from the four equations. Therefore Gaussian elimination (as is used by CRYPTOMINISAT) is strictly weaker than Gauss-Jordan elimination in detecting implications and conflicts.

The efficacy of XOR-handling in the pseudo code of Figure 11 is mainly determined by the procedure *Xorplex*. In essence, two factors, deductive power and computational efficiency, need to be considered in realizing *Xorplex*. We show how the two-literal watching scheme in unit propagation [22] fits incremental Gauss-Jordan elimination in a way similar to the simplex method to support lazy update. Consequently, *Xorplex* can be implemented efficiently and has complete power deducing implications and conflicts whenever they exist.

In the simplex method, the variables of the linear equations $A\mathbf{x} = \mathbf{b}$ are partitioned into m *basic variables* and $(n - m)$ *nonbasic variables* assuming that the $m \times (n + 1)$ matrix $[A|\mathbf{b}]$ is of full rank and $m < n$. Matrix $[A|\mathbf{b}]$ is diagonalized to $[I|A'|\mathbf{b}']$, where I is an $m \times m$ identity matrix and A' is an $m \times (n - m)$ matrix, by Gauss-Jordan elimination such that the m basic and $(n - m)$ nonbasic variables correspond to the columns of I and A' , respectively. Note that diagonalizing $[A|\mathbf{b}]$ to $[I|A'|\mathbf{b}']$ may incur column permutation, which is purely for the ease of visualization to make the columns indexed by the basic variables adjacent to form the identity matrix. In practice, such permutation is unnecessary and not performed. By the simplex method, a basic variable and a nonbasic variable may be interchanged in the process of searching for a feasible solution optimal with respect to some linear objective function. The basic variable to become nonbasic is called the *leaving variable*, and the nonbasic variable to become basic is called the *entering variable*. Although the simplex method was proposed for linear optimization over the reals, the matrix operation mechanism works for our considered XOR-constraints, i.e., linear equations over GF(2).

The problem of XOR-constraint solving is formulated as follows. Given a system of linear equations $A\mathbf{x} = \mathbf{b}$ and a partial truth assignment α to variables $\mathbf{x}' \subseteq \mathbf{x}$, if the induced linear equations $[A|\mathbf{b}]|_{\alpha}$ with respect to α are consistent, derive *all* implications to the non-assigned variables $\mathbf{x} \setminus \mathbf{x}'$. Otherwise, detect a conflicting assignment to \mathbf{x}' that leads to the inconsistency. In fact, Gauss-Jordan elimination achieves this goal as the following proposition asserts.

Proposition 1. *Given a set of XOR-constraints $A\mathbf{x} = \mathbf{b}$ and a partial truth assignment $\alpha : \mathbf{x}' \rightarrow \{0, 1\}$ for $\mathbf{x}' \subseteq \mathbf{x}$, Gauss-Jordan elimination on the induced linear equations $[A|\mathbf{b}]|_{\alpha}$ detects all implications to the non-assigned variables $\mathbf{x} \setminus \mathbf{x}'$ if $[A|\mathbf{b}]|_{\alpha}$ is consistent, or detects a conflict if $[A|\mathbf{b}]|_{\alpha}$ is inconsistent.*

Proof. The proposition follows from the soundness and completeness of GJE for solving a system of linear equations. ■

To equip complete power in deducing implications and conflicts, procedure *Xorplex* of Figure 11 maintains $M|_{\alpha}$ in a reduced row echelon form. Since *Xorplex* is repeatedly applied under various assignments α during SAT solving, Gauss-Jordan elimination needs to be made fast. A two-literal² watching scheme is proposed to make incremental updates on M in a lazy fashion, thus avoiding

² Since the variables in M are of positive phases, there is no need to distinguish “two-literal” or “two-variable” watch.

wasteful computation. Essentially, the following invariant is maintained for M at all times.

Invariant: Given a partial truth assignment α to the variables of matrix $M = [A|b]$, for each row r of M two non-assigned variables are watched. Particularly, the first watched variable (denoted $w_1(r)$) must be a basic variable and the second watched variable (denoted $w_2(r)$) must be a nonbasic variable.

Note that, by this invariant, we assume each row of A contains at least three 1-entries. The reason is that a row without any 1-entry corresponds to either a tautological or conflicting equation, a row with one 1-entry corresponds to an immediate implication, and a row with two 1-entries asserts the equivalence or complementary relation between two variables and is handled specially. Note also that the number of 1-entries in some row of A can possibly be reduced to two later due to incremental Gauss-Jordan elimination. In this situation this row is removed from M and handled specially.

To maintain the invariant, when the two watched variables of some row in M are non-assigned, no action needs to be taken on this row for Gauss-Jordan elimination. On the other hand, actions need to be taken for the following two cases. For the first case, when variable $w_2(r)$ is assigned, another non-assigned nonbasic variable in row r is selected as the new second watched variable. No other rows are affected by this action. For the second case, when $w_1(r)$ is assigned and thus becomes the leaving variable, a non-assigned nonbasic variable in row r needs to be selected as the entering variable. The column c of the entering variable then undergoes the *pivot operation*, which performs row operations (additions) forcing all entries of c to be 0 except for the only 1-entry appearing at row r . Note that the pivot operation may possibly cause the vanishing of variable $w_2(r')$ from another row r' . In this circumstance a new non-assigned nonbasic variable needs to be selected for the second watched variable in row r' , that is, the first case. Note that the process of maintaining the invariant always terminates because for every row r the update of $w_1(r)$ can occur at most once, and thus a row is visited at most m times for M of m rows.

When the invariant can no longer be maintained on some row r of M under α , either of the following two cases happens. Firstly, all variables of r are assigned. In this case the linear equation of r is either satisfied or unsatisfied. For the former, no further action needs to be applied on r ; for the latter, *Xorplex* returns the detected conflict. Secondly, only variable $w_1(r)$ (respectively variable $w_2(r)$) is non-assigned. In this case, the value of $w_1(r)$ (respectively $w_2(r)$) is implied. Accordingly, α is expanded with $w_1(r)$ (respectively $w_2(r)$) assigned to its implied value.

Upon termination, procedure *Xorplex* leads to one of the four results: 1) propagation, 2) conflict, 3) satisfaction, and 4) indetermination. Only the first two cases yield useful information for CDCL SAT solving. The information is provided by procedure *AddXorImplicationConflictClause* in line 15 of the pseudo code in Figure 11. In the propagation case, the corresponding rows in M that implications occur are converted to learnt clauses. In the conflict case, the conflicting row in M is converted to a learnt clause. For example, a propagation

(respectively conflict) occurs at a row corresponding to the linear equation $x_1 \oplus x_2 \oplus x_3 = 0$ under $\alpha(x_1) = 0, \alpha(x_2) = 1$ (respectively $\alpha(x_1) = 0, \alpha(x_2) = 1, \alpha(x_3) = 0$). Then the learnt clause $(x_1 \vee \neg x_2 \vee x_3)$ is produced.

3.2 Implementation Issues

In our actual realization, an $m \times (n + 1)$ matrix M is implemented with a one-dimensional bit array, similar to [26]. Thereby matrix row addition is performed by bitwise XOR operation; a row addition operation translates to n/k bitwise XOR operations, where k is the bit width of a computer word. Moreover, similar to [26], if two XOR-constraint sets have disjoint support variables, they are represented by two individual matrices rather than a single matrix for the sake of memory and computational efficiency.

To support two-literal (or two-variable) watch on M , a watch list is maintained, which provides fast lookup for which rows of M to update when a variable is assigned. To maintain the invariant of two-literal watching, the most costly computation occurs when the basic variable of some row is assigned. It may incur in the worst case $O(m)$ row additions to set a new basic variable for the row. Nevertheless notice that this action cannot make the basic variables of other rows be assigned, and therefore no chain reaction is triggered. For an entire Gauss-Jordan elimination, the time complexity is $O(m^2n)$.

4 Refutation and Interpolation

This section shows how Craig interpolants can be compactly constructed under the framework of *SatSolve*, which combines CDCL-based clause reasoning and GJE-based XOR-constraint solving. Although interpolants for combined propositional and linear arithmetic theories are available under the framework of SMT [20,28,5], they are not directly applicable in our context due to the underlying assumption of most SMT solvers that requires the considered theories to be of disjoint signatures. On the other hand, although theoretically XOR-constraints can always be expressed in CNF and thus propositional interpolation is sufficient, practically such CNF formulae are hard to solve and even if solvable their interpolants can be unreasonably large. A new method awaits to be uncovered.

4.1 Interpolant Generation

For problem formulation, consider interpolant generation for a given unsatisfiable formula $\phi = \phi_A \wedge \phi_B$ with the set V_A of variables of ϕ_A , V_B of ϕ_B , and V_{AB} of common variables shared by ϕ_A and ϕ_B . Let $\phi_A = \varphi_A \wedge \psi_A$ and $\phi_B = \varphi_B \wedge \psi_B$, where φ_A and φ_B are CNF formulae and ψ_A and ψ_B are XOR-constraints. Let M_A (respectively M_B) be the matrix form of the set of linear equations expressed by ψ_A (respectively ψ_B). Then the union of the rows of M_A and M_B corresponds to the matrix M denoted in the previous section.³

³ For an XOR-constraint whose constituent clauses are not all implied by ϕ_A or by ϕ_B , it is not included in M when interpolant derivation is concerned.

If $\varphi_A \wedge \varphi_B$ is already unsatisfiable, the following *clause interpolation rules* [20] suffice to produce the interpolant.

$$\text{CLS-A} \frac{}{C: \langle \bigvee_{l \in C, \text{var}(l) \in V_{AB}} l \rangle} C \in \varphi_A$$

$$\text{CLS-B} \frac{}{C: \langle 1 \rangle} C \in \varphi_B$$

$$\text{CLS-ResA} \frac{C_1 \vee l: \langle I_1 \rangle \quad C_2 \vee \neg l: \langle I_2 \rangle}{C_1 \vee C_2: \langle I_1 \vee I_2 \rangle} \text{var}(l) \in V_A \setminus V_{AB}$$

$$\text{CLS-ResB} \frac{C_1 \vee l: \langle I_1 \rangle \quad C_2 \vee \neg l: \langle I_2 \rangle}{C_1 \vee C_2: \langle I_1 \wedge I_2 \rangle} \text{var}(l) \in V_B$$

Similarly, if $\psi_A \wedge \psi_B$ is already unsatisfiable, the *inequality interpolation rules* [20] suffice for interpolant derivation. They are modified in our context for linear equations over GF(2) in the following.

$$\text{XOR-A} \frac{}{[\mathbf{a}^T | b]: \langle [\mathbf{a}^T | b] \rangle} [\mathbf{a}^T | b] \in M_A$$

$$\text{XOR-B} \frac{}{[\mathbf{a}^T | b]: \langle [\mathbf{0}^T | 0] \rangle} [\mathbf{a}^T | b] \in M_B$$

$$\text{XOR-Sum} \frac{[\mathbf{a}_1^T | b_1]: \langle [\mathbf{a}_1^{*T} | b_1^*] \rangle \quad [\mathbf{a}_2^T | b_2]: \langle [\mathbf{a}_2^{*T} | b_2^*] \rangle}{[\mathbf{a}_1^T | b_1] + [\mathbf{a}_2^T | b_2]: \langle [\mathbf{a}_1^{*T} | b_1^*] + [\mathbf{a}_2^{*T} | b_2^*] \rangle}$$

In the above rules, a partial interpolant, shown in the angle brackets, is associated to each linear equation. Superscript “ T ” and operator “ $+$ ” denote matrix transpose and (modulo 2) matrix addition, respectively. The correctness of these derivation rules is immediate from prior results [20].

Complication arises, however, in interpolant generation when the refutation proof of ϕ involves both clausal resolution and XOR linear arithmetic. Essentially the partial interpolant of a constituent clause of a linear equation is needed. Let C be a constituent clause of equation $\mathbf{a}^T \mathbf{x} = b$, whose partial interpolant is $\mathbf{a}^{*T} \mathbf{x} = b^*$. Then the following derivation rule applies.

$$\text{XORToCLS} \frac{}{C: \langle C^* \vee (\mathbf{a}^{*T} \mathbf{x} = b^*)|_{\neg C} \rangle} C \in [\mathbf{a}^T | b]: \langle [\mathbf{a}^{*T} | b^*] \rangle$$

where $C^* \subseteq C$ with $C^* = \{l \in C \mid \text{var}(l) \in V_{AB} \cap \text{Var}(\mathbf{a}^{*T} \mathbf{x} = b^*)\}$ for $\text{Var}(E)$ denoting the variable set involved in equation E .

Example 3. Consider two equations $[1 \underline{0} \underline{1} 1 1 1] \in M_A$ and $[0 \underline{1} 0 \underline{1} 1 1] \in M_B$ in matrix form over variables $\{x_1, \dots, x_5\}$ with $V_{AB} = \{x_3, x_4, x_5\}$, where the underlined variables are watched. Assume x_1 and x_2 are the basic variables. Under assignment $(x_1 = 0, x_2 = 1)$, the first and second equations are updated to $[1 0 \underline{1} \underline{1} 1 1]$ and $[\underline{1} \underline{1} 1 0 0 0]$, respectively, with new basic variables x_2 and x_4 . The partial interpolant of $[1 1 1 0 0 0]$, i.e., equation $x_1 \oplus x_2 \oplus x_3 = 0$, is derived as follows.

$$\overline{[1\ 0\ 1\ 1\ 1\ 1]: \langle [1\ 0\ 1\ 1\ 1\ 1] \rangle} \quad \overline{[0\ 1\ 0\ 1\ 1\ 1]: \langle [0\ 0\ 0\ 0\ 0\ 0] \rangle}$$

$$[1\ 1\ 1\ 0\ 0\ 0]: \langle [1\ 0\ 1\ 1\ 1\ 1] \rangle$$

Since implication occurs with $x_3 = 1$, a learnt clause $(x_1 \vee \neg x_2 \vee x_3)$ is generated, which is a constituent clause of the clause set $\{(\neg x_1 \vee \neg x_2 \vee \neg x_3), (\neg x_1 \vee x_2 \vee x_3), (x_1 \vee \neg x_2 \vee x_3), (x_1 \vee x_2 \vee \neg x_3)\}$ defined by $x_1 \oplus x_2 \oplus x_3 = 0$. By rule XORToCLS, the partial interpolant of the learnt clause equals

$$\begin{aligned} & x_3 \vee (x_1 \oplus x_3 \oplus x_4 \oplus x_5 = 1) |_{\neg x_1 x_2 - x_3} \\ &= x_3 \vee (x_4 \oplus x_5). \end{aligned}$$

Note that any clause implied by ϕ_A (respectively ϕ_B) can be considered as a clause of ϕ_A (respectively ϕ_B). Similarly any linear equation derivable from M_A (respectively M_B) can be viewed as a linear equation of M_A (respectively M_B). With this observation, one can verify that the partial interpolant derivation for a constituent clause of a linear equation in M_A (respectively M_B) reduces to McMillan's clause interpolation rule for clauses of ϕ_A (respectively ϕ_B). The general correctness of rule XORToCLS is asserted by the following proposition.

Proposition 2. *The partial interpolant derived from rule XORToCLS for $C \in [\mathbf{a}^T | b]$ is consistent with that derived from the clause interpolation rules applied on the clauses clausified from XOR-constraints.*

Proof. Observe that every linear equation $E = [\mathbf{a}^T | b]$ derivable from M can always be expressed as a summation of two equations, one, E_A , derived from a linear combination of equations in M_A and the other, E_B , from M_B . (In fact E_A is the partial interpolant of E .) For C be a constituent clause of E , we show that its partial interpolant derived by XORToCLS is the same as that derived by the clause interpolation rules applied on the resolution sequence leading to C from the clauses of E_A and E_B .

Let the variables appearing in E_A and E_B be divided into five disjoint (possibly empty) subsets: V_1 for those in E_A but not in E_B and V_{AB} , V_2 for those in E_A and V_{AB} but not in E_B , V_3 for those in both E_A and E_B (surely in V_{AB}), V_4 those in E_B and V_{AB} but not in E_A , and V_5 for those in E_B but not in E_A and V_{AB} . Then the variable set of C must be $V_1 \cup V_2 \cup V_4 \cup V_5$.

Because the system consisting of two linear equations $E_A|_{\neg C}$ and $E_B|_{\neg C}$ must be unsatisfiable (due to the fact C being a clause of the summation of E_A and E_B), by the completeness of resolution, C can be derived by resolution on variables V_3 from the clauses of E_A and E_B , more precisely, those clauses whose literals are consistent with C . Since the clauses of E_A and E_B can be considered as clauses in ϕ_A and ϕ_B , respectively, by rule CLS-A the partial interpolants for E_A clauses are subclauses with V_1 variables being removed, and by rule CLS-B the partial interpolants for E_B clauses equal constant 1. Since only V_3 variables are resolved, the partial interpolants are built from pure conjunction operation. As can be verified, the so-derived partial interpolant of C is the same as that of XORToCLS regardless of the detailed resolution steps. ■

In essence rule XORToCLS provides a short cut in generating interpolants. The XOR-constraint reasoning circumvents unnecessary complex fine-grained resolutions and, perhaps more importantly, enforces its equivalent clausal resolution steps being performed within ϕ_A and ϕ_B locally whenever possible. These advantages make compact interpolants derivable from simple generation rules.

4.2 Implementation Issues

For an XOR-equation derived as a summation of rows $R \subseteq M$, its partial interpolant is simply the summation of rows $R \cap M_A$. To derive the partial interpolant, the $m \times (n + 1)$ matrix M is augmented to $M^* = [M|M_A^*]$ by concatenating M with another $m \times n$ matrix M_A^* , which is derived from M by removing the last column and replacing every row belonging to M_B with a row of 0's. Essentially the sub-matrix M_A^* of M^* maintains the partial interpolants at any moment of Gauss-Jordan elimination on M^* . More precisely, in $[M|M_A^*]$, a row in the sub-matrix M_A^* corresponds to the partial interpolant of the same row in the sub-matrix M .

5 Experimental Results

The proposed SAT solving method, named SIMPSAT, was implemented in the C++ language based on CRYPTOMINISAT 2.9.1 (CMS) [26], a state-of-the-art solver equipped with Gaussian elimination. All experiments were conducted on a Linux workstation with a 3.3 GHz Intel Xeon CPU and 64 GB memory. Benchmark examples with many XOR-constraints were taken for experiments.

The first experiment compares our method with CMS on cryptanalysis benchmarks [26]. Four ciphers, Bivium, Trivium, HiTag-2, and Grain, were included with 100 instances each. For fair comparison, same parameters were applied on CMS and SIMPSAT. The results are shown in Table 1 where three methods were applied, namely, CMS⁻ (CMS with GE disabled), CMS⁺ (CMS with GE enabled), and SIMPSAT. The total CPU time averaging over the 100 instances is reported in the second, third, and seventh columns; the portion spent on GE is reported in the fourth and eighth columns; the number of invoked GE calls averaging over the 100 instances is shown in the fifth and ninth columns; the utility of GE, defined as the ratio of the number of useful GE calls (where implication or conflict happened) to that of all GE calls, is listed in the sixth and tenth columns; the speedup of SIMPSAT over CMS⁺ in terms of the average total CPU time (the ratio of that spent by CMS⁺ to that spent by SIMPSAT) is displayed in the eleventh column; the speedup of SIMPSAT over CMS⁺ in terms of the average CPU time taken per GE call (the ratio of that spent by CMS⁺ to that spent by SIMPSAT) is calculated in the last column. To summarize, SIMPSAT exhibited stronger deductive power (as seen by comparing the sixth and tenth columns) in shorter computation time (as seen from the last column) compared with CMS⁺. Thereby SIMPSAT achieved average speedup of 1.69x, 2.00x, 1.21x, and 1.11x for Bivium, Trivium, HiTag-2, and Grain, respectively. Figure 2

Table 1. Performance Comparison on Cryptanalysis Benchmarks

Instance	CMS ⁻	CMS ⁺			SIMPSAT			Spdup over CMS ⁻	GE Spdup per call
	Time (sec)	Time (sec)	Time GE (sec)	#GE	Time (sec)	Time GE (sec)	#GE		
Bivium-45	58.39	65.59	14.70	392200.37	38.39	30.65	9.86	545247.52	63.51
Bivium-46	29.47	26.75	8.09	214578.71	40.99	17.28	5.89	317329.83	64.09
Bivium-47	18.80	17.99	5.79	157721.70	42.39	10.53	4.01	216342.67	63.52
Bivium-48	12.50	11.48	3.91	109732.89	43.74	7.43	2.85	151763.17	66.12
Bivium-49	6.51	6.40	2.70	77970.24	46.91	3.55	1.51	80411.71	66.83
Bivium-50	5.89	4.76	1.97	59077.97	47.25	2.51	1.23	61643.62	67.55
Bivium-51	2.79	2.43	1.13	36940.35	48.48	1.32	0.65	34248.57	67.64
Bivium-52	1.15	1.31	0.66	23139.51	49.88	0.77	0.36	18385.35	68.02
Bivium-53	0.73	0.72	0.40	17602.80	52.45	0.44	0.22	10868.63	69.03
Bivium-54	0.59	0.58	0.27	11318.58	50.98	0.38	0.13	7248.02	68.99
Bivium-55	0.42	0.40	0.22	10905.49	53.43	0.26	0.11	5540.11	69.54
Bivium-56	0.24	0.23	0.12	5958.18	54.16	0.16	0.06	2842.08	71.29
Trivium-151	264.88	2314.04	60.81	1221568.44	36.19	131.14	32.21	17721026.43	60.69
Trivium-152	156.83	140.32	39.95	801775.05	38.31	70.59	19.88	1100015.00	61.63
Trivium-153	72.97	64.18	22.36	437299.75	41.06	30.76	9.91	581075.51	63.13
Trivium-154	57.76	45.57	16.20	316464.91	42.57	20.48	6.50	408162.70	63.38
Trivium-155	31.68	25.90	9.57	190731.45	42.65	13.38	4.57	268314.49	63.09
Trivium-156	15.39	16.72	6.56	133200.84	44.45	8.47	3.06	186959.82	64.14
Trivium-157	15.15	14.56	5.85	124892.01	45.36	7.14	2.63	164411.23	64.66
HiTag2-9	313.58	308.30	2.29	355291.70	7.39	235.89	5.50	1436229.45	22.34
HiTag2-10	146.93	143.32	1.40	208920.52	7.40	115.45	3.18	860728.62	22.13
HiTag2-11	60.87	61.02	0.71	104612.13	7.20	49.63	1.56	425575.32	21.85
HiTag2-12	27.50	27.03	0.40	57723.48	7.49	23.17	0.84	230317.78	21.21
HiTag2-13	14.02	13.63	0.26	38584.40	7.21	11.64	0.48	131037.02	21.31
HiTag2-14	6.24	6.27	0.13	17048.46	6.94	5.37	0.26	68325.91	20.73
HiTag2-15	2.93	2.90	0.07	10649.43	5.77	2.52	0.15	37043.91	20.55
Grain-106	688.50	712.27	35.23	841125.77	8.62	690.27	57.17	13347468.01	30.00
Grain-107	269.72	242.70	17.15	373763.02	8.48	211.73	24.24	1429181.91	29.79
Grain-108	1114.20	119.86	11.41	227777.96	9.33	112.99	14.32	872262.35	31.50
Grain-109	68.83	85.55	8.80	171188.65	9.87	70.54	9.63	592547.87	32.43

compares the performance of SIMPSAT and CMS⁺ on all of the cryptanalysis benchmarks. The CPU times spent by SIMPSAT and CMS⁺ are shown on the *y*-axis and *x*-axis, respectively. As can be seen, SIMPSAT steadily outperformed CMS⁺.

Under a similar setting, experiments were performed on equivalence checking benchmarks for Altera CRC (cyclic redundancy check) circuits [14]. Table 2 compares the performances of ABC cec command [3], CMS⁻, CMS⁺, and SIMPSAT [5]. As can be seen, SIMPSAT is the most robust among the four methods. It is intriguing that SIMPSAT outperforms CMS⁺ by a substantial margin on several examples. Taking the extreme case `crc32-dat48` for example, SIMPSAT finished within 3 seconds while all other methods timed out at 7,200 seconds. A close investigation revealed that SIMPSAT was able to deduce from Gaussian elimination many more powerful short XOR-clauses (with lengths less than or equal to 2) than CMS⁺ as seen from columns six and nine, where the numbers

⁴ A benchmark was prepared by creating a miter structure comparing a design against its synthesized version using a script of ABC commands `dc2`, `dch`, `balance -x`.

⁵ The `cec` command of ABC exploits circuit structure similarities and logic synthesis methods for efficient equivalence checking [18].

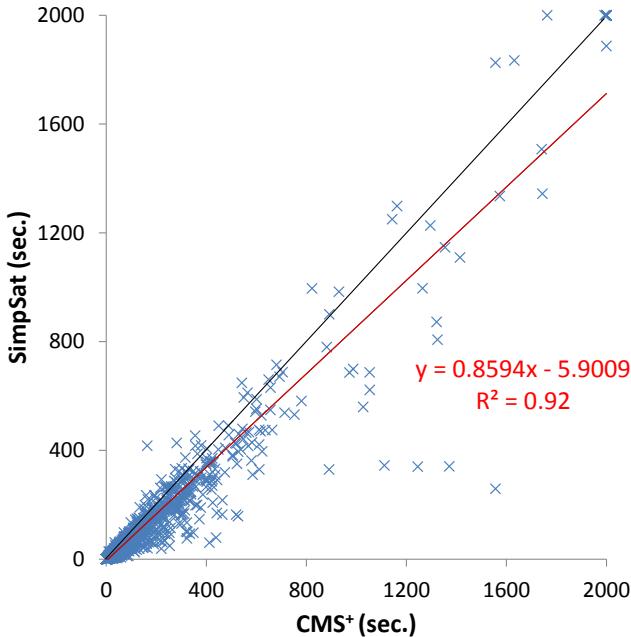


Fig. 2. Runtimes on cryptanalysis benchmarks

of XOR-clauses of lengths less than or equal to 2, denoted “#2xcl,” are shown. These short XOR-clauses contributed to the effectiveness of SIMPSAT.

Another experiment on the benchmarks from randomly generated 3-regular graphs [12] is shown in Table 3. The number of instances of each benchmark suite is shown in the second column; the number of solved instances (within a 7,200-second limit) is shown in the third, fifth, and seventh columns; the entire runtime for solving solvable instances is shown in the fourth, sixth, and eighth columns. SIMPSAT and CMS⁺ achieved similar results.

To study interpolant generation, a prototype, named MINISAT-GE, was built upon MINISAT-p 1.14 [10] (for which proof logging is supported) with XOR-constraint solving integrated as the pseudo code sketched in Figure 1. Benchmarks were created from a subset of the unsatisfiable instances of Table 3 by evenly assigning clauses to ϕ_A and ϕ_B for interpolation. Table 4 compares the interpolants generated from the refutation proofs of MINISAT using McMillan’s clause interpolation rules and those generated from MINISAT-GE using our derivation rules. A 300-second limit was imposed on SAT solving, and interpolants were synthesized using ABC script dc2, dc2, balance. The so generated interpolants were compared in terms of their number of inputs, number of AIG (and-inverter graph) nodes, and number of logic levels. The reported runtime includes SAT solving time and interpolant synthesis time. In the table, an

Table 2. Performance Comparison on Equivalence Checking of CRC Circuits

Instance	ABC cec		CMS ⁻			CMS ⁺			SIMPSAT			Spdup over ABC	Spdup over CMS ⁺
	Time (sec)	Time (sec)	Time (sec)	GE Util (%)	GE #2xcl	Time (sec)	GE Util (%)	GE #2xcl	Time (sec)	GE Util (%)	GE #2xcl		
crc16-dat16	0.11	0.04	0.02	23.37	1	0.02	33.03	15	6.88	1.38	-	-	-
crc16-dat24	0.53	0.17	0.16	26.79	1	0.04	49.53	16	12.93	4.00	-	-	-
crc16-dat32	1.44	1.77	2.05	10.58	2	0.11	33.27	15	12.97	18.48	-	-	-
crc24-dat64	4667.42	>7200	>7200	0.27	0	360.64	3.36	18	12.94	-	-	-	-
crc24-dat64-only-flat	31.52	>7200	>7200	0.17	0	4.71	36.30	22	6.69	-	-	-	-
crc24-zer64-flat	0.62	>7200	29.10	9.39	5	17.13	13.08	11	0.04	1.70	-	-	-
crc24-zer64x2-flat	0.51	498.17	633.49	3.47	0	0.73	16.42	19	0.69	863.20	-	-	-
crc24-zer64x3-flat	0.45	26.84	49.07	0.54	0	0.65	15.31	19	0.70	75.85	-	-	-
crc32c-dat32	596.94	>7200	>7200	22.34	0	0.22	47.95	35	2713.78	-	-	-	-
crc32c-dat64	>7200	>7200	>7200	0.00	7	3386.20	0.14	32	-	-	-	-	-
crc32c-dat64-only	1055.18	>7200	>7200	33.10	3	486.81	20.57	32	2.17	-	-	-	-
crc32c-zer64	0.86	101.39	102.70	4.93	5	0.54	28.78	34	1.59	190.21	-	-	-
crc32c-dat16	0.91	7.21	6.88	10.56	1	0.49	56.18	31	1.85	14.02	-	-	-
crc32c-dat24	2.51	64.94	10.70	30.86	3	0.93	63.11	32	2.71	11.56	-	-	-
crc32c-dat32	385.73	>7200	2153.89	3.38	1	0.49	63.17	32	792.18	4423.46	-	-	-
crc32c-dat40	6666.37	>7200	>7200	6.67	0	0.59	48.28	32	11339.10	-	-	-	-
crc32c-dat48	>7200	>7200	>7200	17.01	4	2.23	57.09	32	-	-	-	-	-
crc32c-dat56	>7200	>7200	>7200	23.36	0	146.22	1.12	32	-	-	-	-	-
crc32c-dat8	0.21	0.40	0.40	0.00	0	0.40	0.00	0	0.52	1.00	-	-	-

Table 3. Performance Comparison on 3-Regular Graph Benchmarks

Instance	#inst	CMS ⁻		CMS ⁺		SIMPSAT	
		#solved	Time (sec)	#solved	Time (sec)	#solved	Time (sec)
mod2-rand3bip-sat	165	103	136064.80	165	6.98	165	7.13
mod2-rand3bip-unsat	75	75	72.46	75	15.65	75	15.66
mod2c-rand3bip-unsat	75	75	962.40	75	871.05	75	862.89
mod2-3cage-unsat	23	23	18.00	23	15.93	23	15.95
mod2c-3cage-unsat	23	23	115.44	23	106.27	23	102.06

entry “-” indicates data unavailable due to timeout, or due to large interpolant sizes not practically synthesizable by ABC. As can be seen, XOR-constraint solving is effective in reducing SAT solving time and admits compact interpolant generation.

6 Related Work

Prior efforts [4, 15, 16, 17] deployed inference rules for XOR-reasoning. In [4], the authors proposed a framework integrating XOR-reasoning with the DPLL procedure using Gauss resolution rules. However there was no implementation provided. In [15], the author focused on recognizing binary and ternary XOR-clauses for equivalence reasoning. Several inference rules were integrated into the DPLL search procedure for literal substitution. Based on the framework of [4], prior work [16, 17] proposed some lightweight inference rules for practical

Table 4. Results on Interpolant Generation

Instance	MINISAT					MINISAT-GE				
	#in	#node	#level	Time SAT (sec)	Time Syn (sec)	#in	#node	#level	Time SAT (sec)	Time Syn (sec)
mod2-rand3bip-unsat-105-1	45	32106	2273	4.32	23.58	45	132	14	0.01	0.1
mod2-rand3bip-unsat-120-1	-	-	-	88.93	-	44	129	12	0.01	0.12
mod2-rand3bip-unsat-135-1	-	-	-	280.31	-	54	159	14	0.01	0.09
mod2-rand3bip-unsat-150-1	-	-	-	53.45	-	50	147	14	0.01	0.09
mod2-rand3bip-unsat-90-1	34	111625	9730	0.63	388.28	34	99	12	0.01	0.09
mod2c-rand3bip-unsat-105-15	-	-	-	63.15	-	57	105	12	0.01	0.1
mod2c-rand3bip-unsat-90-15	30	105500	9375	1.56	175.32	31	5405	538	0.03	25.02
mod2c-3cage-unsat-11	-	-	-	>300	-	70	1857	137	0.01	1.55
mod2-3cage-unsat-9-1	-	-	-	74	-	26	75	12	0.01	0.09
mod2-3cage-unsat-10-1	-	-	-	>300	-	29	84	12	0.01	0.05

XOR-reasoning and supported with conflict-driven learning for XOR-clauses. The DPLL and XOR-reasoning procedures were integrated in a way similar to SMT solvers.

Compared to the closest prior work [26], our approach is similar but with the following main differences. For matrix representation, ours is in a reduced row echelon form, in contrast to the prior row echelon form. For matrix update, ours uses two-variable watching for incremental matrix update, in contrast to the prior column search and row swap. For matrix size, ours maintains a single-sized matrix for propagation/conflict detection, in contrast to the prior doubled-sized matrix. On the other hand, interpolant generation is supported in this work but not previously.

7 Conclusions and Future Work

Boolean satisfiability solving integrated with Gauss-Jordan elimination has been shown powerful in solving hard real-world instances involving XOR-constraints. With two-variable watching and simplex-style matrix update, Gauss-Jordan elimination has been made fast for complete detection of XOR-inferred implications/conflicts. Moreover, Craig interpolation has been made straight for compact interpolant generation, thus bypassing blind and unnecessarily detailed resolutions. For future work, extension to three-variable watching is planned for variable (in)equivalence, in addition to implication and conflict, detection.

Acknowledgments. The authors are grateful to Alan Mishchenko and Sayak Ray for providing the CRC benchmark circuits. This work was supported in part by the National Science Council under grants NSC 99-2221-E-002-214-MY3, 99-2923-E-002-005-MY3, and 100-2923-E-002-008.

References

1. Altera Advanced Synthesis Cookbook,
http://www.altera.com/literature/manual/stx_cookbook.pdf

2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): *Handbook of Satisfiability*. IOS Press (2009)
3. Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, <http://www.eecs.berkeley.edu/~alanmi/abc/>
4. Baumgartner, P., Massacci, F.: The taming of the (X)OR. In: Proc. Int'l Conf. on Computational Logic, pp. 508–522 (2000)
5. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Trans. on Computational Logic* 12(1), 7 (2010)
6. Chen, J.-C.: XORSAT: An efficient algorithm for the DIMACS 32-bit parity problem, CoRR abs/cs/0703006 (2007)
7. Chen, J.-C.: Building a Hybrid SAT Solver via Conflict-Driven, Look-Ahead and XOR Reasoning Techniques. In: Kullmann, O. (ed.) *SAT 2009*. LNCS, vol. 5584, pp. 298–311. Springer, Heidelberg (2009)
8. Craig, W.: Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic* 22(3), 250–268 (1957)
9. Dantzig, G.: Maximization of linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, 339–347 (1951)
10. 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)
11. Gomes, C., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: Proc. National Conf. on Artificial Intelligence (AAAI), pp. 54–61 (2006)
12. Haanpää, H., Jarvisalo, M., Kaski, P., Niemela, I.: Hard satisfiable clause sets for benchmarking equivalence reasoning techniques. *Journal on Satisfiability, Boolean Modeling and Computation* 2(1-4), 27–46 (2006)
13. Jiang, J.-H.R., Lee, C.-C., Mishchenko, A., Huang, C.-Y.: To SAT or not to SAT: Scalable exploration of functional dependency. *IEEE Trans. on Computers* 59(4), 457–467 (2010)
14. Liu, H.-Y., Chou, Y.-C., Lin, C.-H., Jiang, J.-H.R.: Towards Completely Automatic Decoder Synthesis. In: Proc. Int'l Conf. on Computer Aided Design (ICCAD), pp. 389–395 (2011)
15. Li, C.M.: Integrating equivalency reasoning into Davis-Putnam procedure. In: Proc. National Conf. on Artificial Intelligence (AAAI), pp. 291–296 (2000)
16. Laitinen, T., Junttila, T., Niemela, I.: Extending clause learning DPLL with parity reasoning. In: Proc. European Conference on Artificial Intelligence (ECAI), pp. 21–26 (2010)
17. Laitinen, T., Junttila, T., Niemela, I.: Equivalence class based parity reasoning with DPLL(XOR). In: Proc. Int'l Conf. on Tools with Artificial Intelligence (ICTAI), pp. 649–658 (2011)
18. Mishchenko, A., Chatterjee, S., Brayton, R., Een, N.: Improvements to combinatorial equivalence checking. In: Proc. Int'l Conf. on Computer-Aided Design (ICCAD), pp. 836–843 (2006)
19. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) *CAV 2003*. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
20. McMillan, K.L.: An interpolating theorem prover. *Theoretical Computer Science* 345(1), 101–121 (2005)
21. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT-problem: Encoding and analysis. *Journal of Automated Reasoning* 24(1-2), 165–203 (2000)

22. Moskewicz, M., Madigan, C., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. Design Automation Conf. (DAC), pp. 530–535 (2001)
23. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Computeres* 48(5), 506–521 (1999)
24. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM* 53(6), 937–977 (2006)
25. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009)
26. Soos, M.: Enhanced Gaussian elimination in DPLL-based SAT solvers. In: Proc. Pragmatics of SAT (2010)
27. Warners, J., van Maaren, H.: A two-phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters* 23(3-5), 81–88 (1998)
28. Yorsh, G., Musuvathi, M.: A Combination Method for Generating Interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

A Solver for Reachability Modulo Theories

Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri

Microsoft Research

{akashl, qadeer, shuvendu}@microsoft.com

Abstract. Consider a sequential programming language with control flow constructs such as assignments, choice, loops, and procedure calls. We restrict the syntax of expressions in this language to one that can be efficiently decided by a satisfiability-modulo-theories solver. For such a language, we define the problem of deciding whether a program can reach a particular control location as the reachability-modulo-theories problem. This paper describes the architecture of CORRAL, a semi-algorithm for the reachability-modulo-theories problem. CORRAL uses novel algorithms for inlining procedures on demand (Stratified Inlining) and abstraction refinement (Hierarchical Refinement). The paper also presents an evaluation of CORRAL against other related tools. CORRAL consistently outperforms its competitors on most benchmarks.

1 Introduction

The reachability problem, with its roots in the classical theory of finite state machines [20], asks the following question: given a (control flow) graph over a set of nodes and edges, an initial state, and an error state, does there exist a path from the initial to the error state? Subsequent to the recognition that a large class of computer systems can be modeled as finite state machines, this problem received a lot of attention from researchers interested in formal verification of computer systems [19][7]. Over the years, many variations of this problem have been proposed to model increasingly complex systems. For example, finite control is augmented by a stack to model procedure calls in imperative programs or by a queue to model message passing in concurrent programs. Along a different dimension, researchers have proposed annotating the nodes and edges of the graph by a finite alphabet to enable specification of temporal behavior [30].

We are concerned with the problem of reasoning about programs written in real-world imperative programming languages such as C, C#, and Java. Because such programs routinely use unbounded data values such as integers and the program heap, the framework of finite-state machines is inadequate for modeling them. We propose that the semantic gap between the programming languages and the intermediate modeling and verification language should be reduced by allowing modeling constructs such as uninterpreted functions and program variables with potentially unbounded values, such as integers, arrays, and algebraic datatypes. We refer to the reachability problem on such a modeling language as *reachability-modulo-theories*. Even though this generalization

immediately leads to a reachability problem that is undecidable, we feel that this direction is a promising one for the following reasons. First, the increased expressiveness dramatically simplifies the task of translating imperative software to the intermediate language, thus making it much easier to quickly implement translators from languages. In our own work, we have developed high-fidelity translators both for C and .NET bytecode with a relatively modest amount of engineering effort. The presence of an intermediate modeling and verification language simplifies the construction of end-to-end verification systems by decoupling the problem of model construction from the problem of solving reachability queries on the model. Second, our generalization leads to an intermediate verification language whose expression language is expressible in the framework of satisfiability-modulo-theories (SMT) and decidable efficiently using the advanced solvers [15][12] developed for this framework. Therefore, reachability on bounded program fragments can be decided by converting them into verification conditions [14][4]. The ability to decide bounded reachability in a scalable fashion can be of tremendous value in automated bug-finding and debugging.

This paper describes CORRAL, a solver for a restricted version of the reachability-modulo-theories problem, in which the depth of recursion is bounded by a user-supplied *recursion bound* (we assume that loops are converted to recursive procedures).¹ As discussed earlier, recursion-bounded reachability-modulo-theories is decidable if the expression language of programs is decidable. In fact, the simplest method to solve this problem is to statically inline all procedures up to the recursion bound, convert the resulting program into a verification condition (VC), and present the VC to an SMT solver. While this approach may work for small programs, it is unlikely to scale because the inlined program may be exponentially large. To make this point concrete, consider the restricted case of recursion-free (and loop-free) programs. For these programs, a recursion bound of 0 suffices for full verification; However, it is still a nontrivial problem to solve because the inlined program could be exponential in the size of the original program. This complexity is fundamental, in fact, the reachability-modulo-theories for recursion-free programs becomes PSPACE-hard even if we allow just propositional variables. If we add other theories such as uninterpreted functions, arithmetic, or arrays that are decidable in NP, reachability-modulo-theories for recursion-free programs is decidable in NEXPTIME; however, we conjecture that the problem is NEXPTIME-hard. Since the efficient (in practice) subset of theories decided by an SMT solver are in the complexity class NP, it is likely that that the exponential complexity of inlining is unavoidable in the worst case; CORRAL provides a solution to avoid (or delay) this exponential complexity.

1.1 The Corral Architecture

CORRAL embodies a principled approach to tackling the complexity of recursion-bounded reachability-modulo-theories (RMT). Its overall architecture is shown

¹ CORRAL can also be used in a loop where the recursion bound is increased iteratively, in which case it is a semi-algorithm to the reachability-modulo-theories problem.

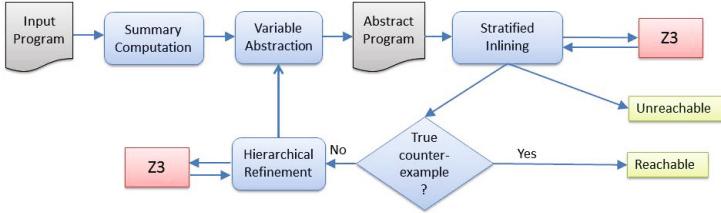


Fig. 1. CORRAL’s architecture

in Figure 11 CORRAL uses BOOGIE [3] as the modeling language for encoding reachability queries. The BOOGIE language already supports the essential requirements of RMT: it has the usual control-flow constructs (branches, loops, procedures), unbounded types (integers and maps) and operations such as arithmetic and uninterpreted functions. Moreover, the BOOGIE framework [3] comes equipped with verification-condition generation algorithms that can convert a call-free fragment of code with an assertion to an SMT formula such that the latter is satisfiable if and only if there is an execution of the code that violates the assertion. This formula can then be fed to various solvers, including Z3 [12]. Thus, the BOOGIE framework can solve the loop-free call-free segment of RMT.

CORRAL has a two-level counterexample-guided abstraction refinement [9] (CEGAR) loop. The top level loop performs a localized abstraction over global variables [9,23]. Given a set of tracked variables T , it abstracts the input program using T ; the initial set of tracked variables is empty. Next, it feeds the abstracted program to the module denoted *Stratified Inlining* (§2) to look for a counterexample. Since VC generation is quadratic in the number of program variables, performing variable abstraction before stratified inlining can improve the latter’s performance significantly. If stratified inlining finds a counterexample, it is checked against the entire set of global variables. If the path is feasible, then it is a true bug; otherwise, we call the module denoted *Hierarchical Refinement* (§3.1) to minimally increase the set of tracked variables, and then repeat the process. Besides the outer loop that increases T , both the stratified inlining and hierarchical refinement algorithms have their own iterative loops that require multiple calls to Z3.

Stratified Inlining. For a single procedure program, we simply generate its VC and feed it to Z3. In the presence of multiple procedures, instead of inlining all of them up to a given bound, we only inline a few, generate its VC and ask Z3 to decide reachability. If it finds a counterexample, then we’re done. Otherwise, we replace every non-inlined call site with a summary of the called procedure. This results in a VC that over-approximates the program, which is fed to Z3. A counterexample in this case (if any) tells us which procedures to inline next. By default, CORRAL uses a summary that havoccs all variables potentially modified by the procedure, but more precise summaries can be obtained from any static analysis. In our experiments, we used HOUDINI [17] to compute summaries.

Hierarchical Refinement. This algorithm takes a divide-and-conquer approach to the problem of discovering a minimal set of variables needed to refute an infeasible counterexample. Suppose n is the total number of variables. In the common case when the number of additional variables needed to be tracked is small compared to n , our algorithm makes only $O(\log(n))$ path queries to Z3, as compared to a previous algorithm [23] that makes $O(n)$ queries.

We have evaluated CORRAL on a large collection of benchmarks to measure its robustness and performance (§4). To demonstrate robustness, we ran CORRAL on programs obtained from multiple sources; they are either sequential C programs, or concurrent C programs sequentialized using the Lal-Reps algorithm [25]. To demonstrate good performance, we compare against existing methods: hierarchical refinement is compared against the STORM refinement algorithm, stratified inlining is compared against static inlining, and the entire CORRAL system is compared against a variety of software verification tools such as SLAM [2], YOGI [29], CBMC [8], and ESBMC [11]. Our evaluation shows that CORRAL performs significantly better than existing tools. Moreover, every tool other than CORRAL had a difficult time on some benchmark suite (i.e., the tool would run out of time or memory on most programs in that suite).

Contributions. In summary, this paper makes the following contributions:

- The stratified inlining algorithm.
- The hierarchical refinement algorithm for refining variable abstraction.
- The design and implementation of CORRAL, a novel architecture that combines summaries, variable abstraction, and stratified inlining to solve the reachability-modulo-theories problem.
- Extensive experimental evaluation to demonstrate the robustness and performance of CORRAL.

2 Stratified Inlining

We consider a simple imperative programming language in which each program has a vector of global variables denoted by g and a collection of procedures, each of which has a vector of input parameters denoted i and a vector of output parameters denoted by o . Given such a program F (possibly with recursion, but no loops), a procedure \mathbf{m} in the program, an initial condition $\varphi(i, g)$ over i and g , and a final condition $\psi(o, g)$ over o and g , the goal is to determine whether there is an execution of \mathbf{m} that begins in a state satisfying φ and ends in a state satisfying ψ . The pseudo-code of our algorithm is shown in Fig. 2.

Stratified inlining uses under- and over-approximation of procedure behaviors to inline procedures on demand. To underapproximate a procedure, we simply block all executions through it, i.e., we replace it with “assume false”. To overapproximate a procedure, we use a *summary* of the procedure, i.e., a valid over-approximation of the procedure’s behaviors. The default summary of a procedure havoc all variables potentially modified by it and leaves the output value completely unconstrained. One may use any static analysis to compute better

summaries. In some of our experiments (§4.2), we used HOUDINI [17] to compute summaries, which improved the performance of CORRAL over using the default summaries. We call the overapproximation of a procedure p as $\text{summary}(p)$.

The algorithm maintains a partially-inlined program P starting from the procedure m (line 1). It also maintains the set C of non-inlined (or *open*) call-sites in P . A call-site is defined as a dynamic instance of a procedure call, i.e., the procedure call along with the runtime stack of unfinished calls. For instance, the call-site $c = [m; \text{foo1}; \text{foo2}]$ refers to a call to `foo2` from `foo1`, which in turn was called from `m`.

We define the *cost* of a call-site c to be the number of occurrences of the top-most procedure in c . For instance, if $c = [m; \text{foo1}; \text{foo2}; \text{foo1}; \text{foo2}; \text{foo1}]$ then $\text{cost}(c) = 3$ because `foo1` occurs thrice in c . In other words, the cost of a call-site c reflects the number of recursive calls necessary to reach c . Given a set C of call-sites, the function $\text{split-on-cost}(C, r)$ partitions C into two disjoint sets C_1 and C_2 , where the latter has all those call-sites whose cost is r or greater.

```

Input: Program  $F$  and its starting procedure  $m$ 
Input: An initial condition  $\varphi$  and a final condition  $\psi$ 
Input: Maximum recursion depth  $\text{MAX}$ 
Output: CORRECT,  $\text{BUG}(\tau)$ , or NoBUGFOUND
1:  $P := \{\text{assume } \varphi; m; \text{assert } \neg\psi\}$ 
2:  $C := \text{open-call-sites}(P)$ 
3: for  $r = 1$  to  $\text{MAX}$  do
4:   while true do
5:     //Query-type 1
6:      $P' = P[\forall_{c \in C} c \leftarrow \text{assume false}]$ 
7:     if  $\text{check}(P') == \text{BUG}(\tau)$  then
8:       return  $\text{BUG}(\tau)$ 
9:      $C_1, C_2 := \text{split-on-cost}(C, r)$ 
10:    //Query-type 2
11:     $P' = P[\forall_{c \in C_1} c \leftarrow \text{summary}(c),$ 
 $\forall_{c \in C_2} c \leftarrow \text{assume false}]$ 
12:     $\text{ret} := \text{check}(P')$ 
13:    if  $\text{ret} == \text{CORRECT} \wedge C_2 == \emptyset$  then
14:      return CORRECT
15:    if  $\text{ret} == \text{CORRECT} \wedge C_2 \neq \emptyset$  then
16:      break
17:    let  $\text{BUG}(\tau) = \text{ret}$ 
18:     $P := P + \{\text{inline}(c) \mid c \in C, c \in \tau\}$ 
19:     $C := \text{open-call-sites}(P)$ 
20: return NoBUGFOUND

```

Fig. 2. The stratified inlining algorithm

cause their cost is r or more. If the resulting program is correct and C_2 was empty, then all open call-sites were overapproximated. Thus, the original program F has no bugs (line 14). If C_2 was not empty, then r is not sufficient to conclude any answer, thus we break to line 3 and increment r . If the check on line 11 found a trace τ , then τ must pass through some call-sites in C (because line 8 was not taken). All open calls on τ are inlined and the algorithm repeats.

Each iteration of the loop at line 3 looks for a bug (a valid execution of m that violates ψ) within the cost r . The loop at line 4 inlines procedures on-demand. Each of its iterations comprise of two main steps. In the first step, each open call-site in P is replaced by its underapproximation (line 6) to obtain a closed program P' , which is checked using a theorem prover (line 7). The routine *check* takes a bounded program as input and feeds it to the theorem prover to determine satisfiability of the assertion in the program. If a satisfying path is found, the algorithm terminates with `BUG` (line 8).

The second step involves overapproximation. Line 11 replaces each call-site c that has not reached the bound r (i.e., $c \in C_1$) with the summary of the called procedure. Other call-sites $c \in C_2$ are still blocked because their cost is r or more. If the resulting program is correct and C_2 was empty, then all open call-sites were overapproximated. Thus, the original program F has no bugs (line 14). If C_2 was not empty, then r is not sufficient to conclude any answer, thus we break to line 3 and increment r . If the check on line 11 found a trace τ , then τ must pass through some call-sites in C (because line 8 was not taken). All open calls on τ are inlined and the algorithm repeats.

Iteratively increasing the recursion bound ensures that in the limit (when MAX approaches ∞), stratified inlining is complete for finding bugs.

The main advantages of the stratified inlining algorithm are:

1. The program provided to the automated theorem prover is generated incrementally. Eager static inlining (which inlines all procedures upfront up to the recursion bound) creates an exponential explosion in the size of the program. Stratified inlining delays this exponential explosion.
2. Stratified inlining inlines only those procedures that are relevant to ruling out spurious counterexamples. Thus, it can often perform the search while inlining few procedures. This ability makes the search property-guided. Because of the use of over-approximations, stratified inlining can be faster than static inlining even for correct programs.
3. If the program is buggy, then a bug will eventually be found, assuming that the theorem prover always terminates and $\text{MAX} = \infty$.

3 Variable Abstraction and Refinement

It is often the case that a majority of program variables are not needed for proving or disproving reachability of a particular goal. In this case, abstracting away such variables can help the stratified inlining algorithm because: (1) The VC construction is quadratic in the number of variables [4]; and (2) the theorem prover does not get distracted by irrelevant variables. However, abstraction can lead to an over-approximation of the original program and lead to spurious counterexamples. In this case, we use a refinement procedure to bring back some of the variables that were removed.

We apply variable abstraction to only global variables. At any point in time, the set of global variables that are retained by the abstraction are called *tracked* variables. Note that BOOGIE programs always have a finite number of global variables; unbounded structures in real programs such as the heap are modeled using a finite number of maps. Thus, the refinement loop always terminates.

Variable abstraction, also known as *localization reduction* [9], has been used extensively in hardware verification. We now briefly describe the variable abstraction algorithm implemented in CORRAL (Fig. 3); this algorithm was previously implemented in the STORM checker [23]. Later, we present a new refinement algorithm called *hierarchical refinement* (§3.1) that significantly out-performs the refinement algorithm in STORM [23].

Let *IsGlobalVar* be a predicate that is *true* only for global variables. Let *GlobalVars* be a function that maps an expression to the set of global variables that appear in that expression. Let T be the current set of tracked variables. Variable abstraction is carried out as a program transformation, applied statement-by-statement, as shown in Fig. 3. Essentially, assignments to variables that are not tracked are completely removed (replaced by “assume true”). Further, expressions that have an untracked global variable are assumed to evaluate to any value. Consequently, assignments whose right-hand side have such an expression are converted to non-deterministic assignments (havoc statements).

$v := e$ $\mapsto \begin{cases} \text{assume true} & IsGlobalVar(v) \wedge v \notin T \\ \text{havoc } v & GlobalVars(e) \not\subseteq T \\ v := e & \text{otherwise} \end{cases}$	$\text{assume } e$ $\mapsto \begin{cases} \text{assume true} & GlobalVars(e) \not\subseteq T \\ \text{assume } e & \text{otherwise} \end{cases}$
$\text{assert } e$ $\mapsto \begin{cases} \text{assert false} & GlobalVars(e) \not\subseteq T \\ \text{assert } e & \text{otherwise} \end{cases}$	$\text{havoc } v$ $\mapsto \begin{cases} \text{assume true} & v \notin T \\ \text{havoc } v & \text{otherwise} \end{cases}$

Fig. 3. Program transformation for variable abstraction, with tracked variables T

3.1 Hierarchical Refinement

In CORRAL, we abstract the program using variable abstraction and then feed it to the stratified inlining routine. If it returns a counterexample, say τ , which exhibits reachability in the abstract program, then we check to see if τ is feasible in the original program by *concretizing* it, i.e., we find the corresponding path in the input program. If this path is infeasible, then τ is a spurious counterexample. The goal of refinement is to figure out a *minimal* set of variables to track that rule out the spurious counterexample.² STORM’s refinement algorithm already computes a minimal set, but the algorithms presented in this section find such a set much faster.

Let us define two subroutines: *Abstract* takes a path and a set of tracked variables and carries out variable abstraction on the path. The subroutine *check* takes a path (with an assert) and returns **CORRECT** only when the assert cannot be violated on the path. We implement *check* by simply generating the VC of the entire path and feeding it to Z3.

Let G be the entire set of global variables and T be the current set of tracked variables. STORM’s refinement algorithm requires about $|G - T|$ number of iterations, and each iteration requires at least one call to *check*. In our experience, we have found that most spurious counterexamples can be ruled out by tracking one or two additional variables. We leverage this insight to design faster algorithms.

Alg. ② uses a divide-and-conquer strategy. It has best-case running time when only a few additional variables need to be tracked, in which case the algorithm requires $\log(|G - T|)$ number of calls to *check*. In its worst case, which happens when all variables need to be tracked, it requires at most $2|G - T|$ number of calls to *check*. As our experiments show, most refinement queries tend to be towards the best case, which is an exponential improvement over STORM.

Alg. ② uses a recursive procedure *hrefine* that takes three inputs: the set of tracked variables (T), the set of do-not-track variables (D , initially empty), and a path P . It assumes that P , when abstracted with $G - D$, is correct (i.e., assertion in *Abstract*($P, G - D$) cannot be violated). We refer to Alg. ② as the top-level call *hrefine*(T, \emptyset, P). Note that while *hrefine* is running, the arguments T and D can change across recursive calls, but P remains fixed to be the input counterexample.

² We do not attempt to find the *smallest* set of variables to track; not only might that be very hard to compute, but a minimal set already gives good overall performance for CORRAL.

Procedure <i>hrefine</i> (T, D, P) Input: A correct path P with global variables G Input: A set of variables $T \subseteq G$ that must be tracked Input: A set of variables $D \subseteq G$ that definitely need not be tracked, $D \cap T = \emptyset$ Output: The new set of variables to track 1: if $T \cup D = G$ then 2: return T 3: $P' := \text{Abstract}(P, T)$ 4: if $\text{check}(P') == \text{CORRECT}$ then 5: return T 6: if $ G - (T \cup D) = 1$ then 7: return $G - D$ 8: $T_1, T_2 := \text{partition}(G - (T \cup D))$ 9: $S_1 := \text{hrefine}(T \cup T_2, D, P)$ 10: $S'_1 := S_1 \cap T_1$ 11: return $\text{hrefine}(T \cup S'_1, D \cup (T_1 - S'_1), P)$	Procedure <i>vcrefine</i> (T, D, P) Input: A program P with special Boolean constants B Input: A set of Boolean constants $T \subseteq B$ that must be set to <i>true</i> Input: A set of Boolean constants $D \subseteq B$ that must be set to <i>false</i> Output: The set of Boolean constants to set to <i>true</i> 1: if $T \cup D = B$ then 2: return T 3: $\psi = (\bigwedge_{b \in T} b) \wedge (\bigwedge_{b \in B - T} \neg b)$ 4: if $\text{check}(\text{VC}(P) \wedge \psi) == \text{CORRECT}$ then 5: return T 6: if $ B - (T \cup D) = 1$ then 7: return $B - D$ 8: $T_1, T_2 := \text{partition}(B - (T \cup D))$ 9: $S_1 := \text{vcrefine}(T \cup T_2, D, P)$ 10: $S'_1 := S_1 \cap T_1$ 11: return $\text{vcrefine}(T \cup S'_1, D \cup (T_1 - S'_1), P)$
(a) Algorithm 1: <i>hrefine</i> (T, \emptyset, P)	(b) Algorithm 2: <i>vcrefine</i> (T, \emptyset, P)

Fig. 4. Hierarchical refinement algorithms

Alg. 1 works as follows. If $T \cup D = G$ (line 1) then we already know that P is correct while tracking T (because the precondition is that P is correct while tracking $G - D$). Lines 3 to 5 check if T is already sufficient. Otherwise, in line 6, we check if only one variable remains undecided, i.e., $|G - (T \cup D)| = 1$, in which case the minimal solution is to include that variable in T (which is the same as returning $G - D$). Lines 8 to 11 form the interesting part of the algorithm. Line 8 splits the set of undecided variables into two equal parts randomly. Because of the check on line 6, we know that each of T_1 and T_2 is non-empty. Next, the idea is to use two separate queries to find the set of variables in T_1 (respectively, T_2) that should be tracked. The first query is made on line 9, which tracks all variables in T_2 . The only remaining undecided variables for this query is the set T_1 . Thus, the answer S_1 of this query will include $T \cup T_2$ along with the minimal set of variables in T_1 that should be tracked. We capture this in S'_1 and then all variables in $T_1 - S'_1$ should not be tracked. Thus, the second query includes S'_1 in the set of tracked variables and $(T_1 - S'_1)$ in the set of do-not-track variables. The procedure *hrefine* is guaranteed to return a minimal set of variables to track.

Theorem 1. *Given a path P with global variables G , and sets $T, D \subseteq G$, such that T and D are disjoint, suppose that $\text{Abstract}(P, G - D)$ is correct. If $R = \text{hrefine}(T, D, P)$ then $T \subseteq R \subseteq G - D$, and $\text{Abstract}(P, R)$ is correct, while for each set R' such that $T \subseteq R' \subset R$, $\text{Abstract}(P, R')$ is buggy.*

Proof of this theorem can be found in our techreport [24]. The following Lemma describes the running time of the algorithm.

Lemma 1. *If the output of Alg. 1 is a set R , then the number of calls to *check* made by the algorithm is (a) $O(|R - T| \log(|G - T|))$, and (b) bounded above by $\max(2|G - T| - 1, 0)$.*

```

if( $\neg \text{Tracked}(v)$ ) {
    assume true;
} else if( $\neg \text{Tracked}(e)$ ) {
    havoc v;
} else {
    v := e;
}

```

(a)


```

if( $\neg \text{Tracked}(e)$ ) {
    assume true;
} else {
    assume e;
}

```

(b)


```

if( $\neg \text{Tracked}(e)$ ) {
    assert false;
} else {
    assert e;
}

```

(c)

Fig. 5. Program transformation for parameterized variable abstraction: (a) Transformation for $v := e$; (b) assume e ; (c) assert e . Other statements are left unchanged.

Both Alg. 1 and STORM’s refinement share a disadvantage: they spend a significant amount of time outside the theorem prover. Each iteration of Alg. 1 needs to abstract the path with a different set of variables, and then generate the VC for that path. In order to remove this overhead, the next algorithm that we present will require VC generation only once and the refinement loop will be carried out inside the theorem prover.

First, we carry out a *parameterized* variable abstraction of the input path as follows: for each global variable v , we introduce a Boolean constant b_v and carry out the program transformation shown in Fig. 5. The transformed program has the invariant: if b_v is set to *true* then the program behaves as if v is tracked, otherwise it behaves as if v is not tracked. The transformation uses a subroutine *Tracked*, which takes an expression e as input and returns a Boolean formula:

$$\text{Tracked}(e) = \bigwedge_{v \in \text{GlobalVars}(e)} b_v$$

Thus, *Tracked*(e) returns the condition under which e is tracked.

If P_I was the input counterexample and T the current set of tracked variables, then we transform P_I to P using parameterized variable abstraction. Next, we set each b_v , $v \in T$ to *true*. Let B be the set of Boolean constants b_v , $v \notin T$. The refinement question now reduces to: what is the minimum number of Boolean constants in B that must be set to *true* so that P is correct, given that setting all constants in B to *true* makes P correct? We solve this using Alg. 2, which takes P as input.

Alg. 2 is exactly the same as Alg. 1 with the difference that instead of operating at the level of programs and program variables, it operates at the level of formulae and Boolean constants. This buys us a further advantage: the queries made to the theorem prover on line 6 are very similar. One can save $\text{VC}(P)$ on the theorem prover stack and only supply ψ for the different queries. This enables the theorem prover to reuse all work done on $\text{VC}(P)$ across queries.

4 Evaluation

We have implemented CORRAL as a verifier for programs written in BOOGIE. It is supported by a front-end each for compiling C and .NET bytecode to BOOGIE. The translation from C to BOOGIE uses several theories for encoding the semantics of C programs. Linear arithmetic is used for modeling pointer arithmetic and a subset of integer operations; uninterpreted functions are used for modeling any other operation not modeled by linear arithmetic; arrays are used to

Name	LOC	Vars	Procs	Conc?	Correct?	Iter	Time (sec)		
							Total	R(%)	S(%)
daytona	660	114	40	Yes	Yes	8	26.9	50	35
daytona_bug2	660	114	40	Yes	No	6	27.0	56	27
kbdclass_read	978	212	48	Yes	Yes	12	194.4	52	29
kbdclass_ioctl	978	212	48	Yes	Yes	6	63.9	43	38
mouclass_read	818	179	44	Yes	Yes	13	185.7	53	28
mouclass_bug3	818	179	44	Yes	No	15	245.5	53	30
ndisprot_write	907	122	46	Yes	Yes	6	24.8	41	44
pcidrv_bug1	661	109	49	No	No	11	37.4	49	37
serial_read	1601	378	77	Yes	Yes	13	1151.7	41	51
mouser_sdv_a	3311	225	131	No	No	4	35.4	33	45
mouser_sdv_b	3898	252	143	No	Yes	12	990.8	15	81
fdc_sdv	5799	421	180	No	Yes	11	659.8	11	85
serial_sdv_a	7373	466	149	No	Yes	6	139.2	47	34
serial_sdv_b	7396	439	168	No	No	6	289.1	46	40

Fig. 6. Running times of CORRAL on driver benchmarks

model the heap memory split into multiple maps based on fields and types [10]. The translation of .NET bytecode uses the aforementioned theories in a similar way but also uses two other theories: (1) algebraic datatypes to model object types and delegate values, and (2) generalized array theory [13] to model hashsets and .NET events. It is important to note that CORRAL is agnostic to the source language used and depends only on the compiled BOOGIE program.

4.1 Evaluating Components of Corral

The first set of experiments show: (1) how the various components of CORRAL contribute to its overall running time; (2) a comparison of stratified inlining against static inlining; and (3) a comparison of different refinement algorithms. These experiments were conducted on a collection of Windows device drivers in C (compiled to BOOGIE using HAVOC [10]). For each driver, we had various different harnesses that tested different functionalities of the driver. Some of the harnesses were concurrent, in which case the sequential program was obtained using a concurrent-to-sequential source transformation described elsewhere [16][25]. Such sequentialized programs are often quite complicated (because they simulate a limited amount of concurrency using non-determinism in data) and form a good test bench for CORRAL. These drivers also had planted bugs denoted by the suffix “bug” in the name of the driver.

A summary of these drivers and CORRAL’s running time is shown in Fig. 6. We report: the number of non-empty C source lines (LOC), the number of global variables in the generated BOOGIE file (Vars), the number of procedures (Procs), whether the driver is concurrent (Conc?), whether it has a planted bug or not (Correct), the number of iterations of the refinement loop (Iter), the total running time of CORRAL (Total), the fraction of time spent in abstraction and refinement (R%) and the fraction of time spent in checking using the stratified inlining algorithm (S%). The refinement algorithm used was Alg. 2. CORRAL fares reasonably well on these programs. A significant fraction of the time is spent refining, thus, justifying our investment into faster refinement algorithms.

Static Inlining. For the above-described run of CORRAL, we collected all programs that were fed to stratified inlining and ran static inlining on them, i.e., we inlined all procedures (up to the recursion bound) upfront and then fed it to Z3. This did not work very well: static inlining ran out of memory (while creating the VC) on each of the six largest programs, with a recursion bound of 1 or 2. Even when the VC did fit in memory, Z3 timed out (after 1 hour) in many cases. For the rest, stratified inlining was still three times faster. A more detailed comparison can be found in our techreport [24].

Hierarchical Refinement. For the above-described run of CORRAL, we collected all the spurious counterexamples and ran each of the three refinement algorithms on them: STORM’s refinement algorithm, Alg. 1 and Alg. 2. All three algorithms returned the same answer on each counterexample. Alg. 2 was a clear winner with very little variation in its running time. The average speedup of Alg. 2 over Alg. 1 was 2.8X and that over STORM was 13.2X. A more detailed comparison can be found in our techreport [24].

4.2 SDV Benchmarks

For our next experiment, we compare CORRAL against state-of-the-art verification tools SLAM [2] and YOGI [29]. Both these tools are run routinely by the Static Driver Verifier (SDV) team against a comprehensive regression suite consisting of multiple (real) drivers and properties. The suite consists of a total of 2274 driver-property pairs, of which 1886 are correct (i.e., the property is satisfied by the driver) and 388 are incorrect (i.e., there is an execution of the driver that violates the property). We compare CORRAL against YOGI. (The comparison against SLAM is similar in nature.) We compare the running times of the tools as well as evaluate the effectiveness of recursion bounding. We will use the term program to refer to a driver instrumented with a particular property.

First, note that YOGI does full verification unlike CORRAL that requires a recursion bound to cut-off search. Thus, YOGI has to do more work than CORRAL for correct programs. Moreover, both these tools use slightly different modeling of C semantics, leading to different answers for a few programs. It was difficult to remove these differences.

We ran CORRAL with a recursion bound of 2. The natural question to ask is: how many bugs did CORRAL miss? It turns out that only on 9 programs YOGI found a bug, but CORRAL was not able to find one (out of a total of 388 buggy programs). We investigated these 9 programs and they turned out to be instances of just two loops. The first loop required a recursion bound of 3, and CORRAL was able to find the bugs with this bound. The second loop was of the form **for**($i = 0; i < 27; i++$), and thus, required a bound of 27 for CORRAL to explore the code that came after this loop. We are currently investigating techniques to deal with such constant-bounded loops.

Aggregating over all programs on which YOGI and CORRAL returned the same answer, YOGI took 55K seconds to produce an answer for all of them, whereas CORRAL required only 27K seconds, a speedup of about 2X. The scatter

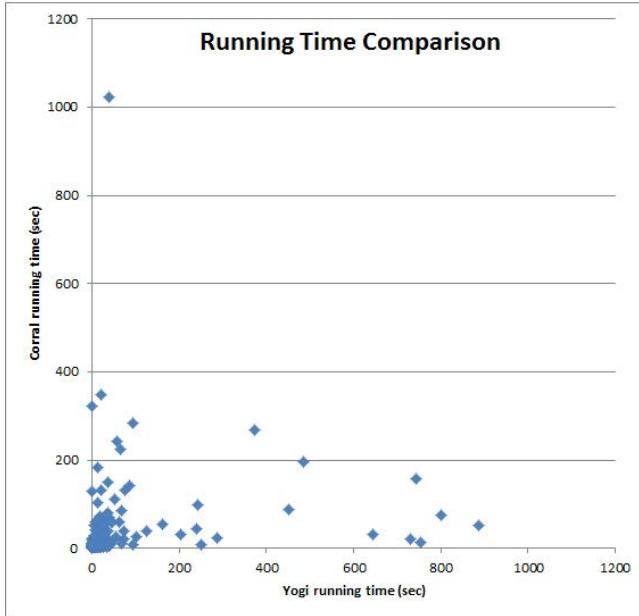


Fig. 7. A comparison of running times of CORRAL against YOGI on buggy programs

plot comparing the running times of YOGI and CORRAL on buggy programs is shown in Fig. 7. Both tools took around the same amount of total time on these programs (about 10K seconds). However, the total running time is dominated by many programs with a trivial running time, on which CORRAL is slightly slower. The distribution of points in the scatter plot show a more interesting trend. For instance, there is a larger distribution of points around the X-axis, meaning that CORRAL did very well on programs on which YOGI was having a hard time. Aggregating over buggy instances on which YOGI took at least a minute, CORRAL was twice as fast as YOGI. There is also a smaller distribution of points around the Y-axis on which YOGI did much better. Manual inspection of these programs revealed that the main reason for YOGI’s success was summarization of procedures using predicate abstraction. The next subsection equips CORRAL with a simple summarization routine.

We tried using other bounded-model checkers (CBMC [8] and ESBMC [11]) on these programs but they did not perform well. They either ran out of time or memory on most programs, possibly because they use static inlining to deal with procedures. Moreover, both SLAM and YOGI do not work well on “sequentialized” versions of concurrent programs (i.e., they would often time out). On the other hand, CORRAL is able to uniformly work on all these programs.

In conclusion, this experiment shows: (1) the practical applicability of recursion bounding in practice for real-world bug hunting, and (2) that CORRAL is competitive with state-of-the-art software model checkers.

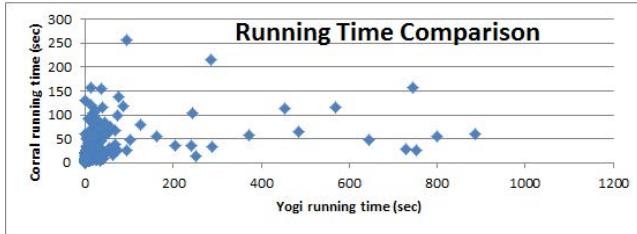


Fig. 8. A comparison of running times of CORRAL augmented with HOUDINI against YOGI on buggy programs

Using Summaries. We now demonstrate the ability of CORRAL to leverage abstraction techniques by using HOUDINI [17]. HOUDINI is a scalable modular analysis for finding the largest set of inductive invariants from amongst a user-supplied pool of candidate invariants. HOUDINI works as a fixpoint procedure; starting with the entire set of candidate invariants, it tries to prove that the current candidate set is inductive. The candidates that cannot be proved are dropped and the procedure is repeated until a fixpoint is reached. Our usage of HOUDINI is restricted to inferring post-conditions of procedures, where a post-condition is simply a predicate on the input and output vocabulary of a procedure. The inferred post-conditions act as procedure summaries.

The drivers in our suite do not have recursion (but may have loops). In this setting, if we generate n candidate summaries per procedure, and there are P procedures, then HOUDINI requires at most $O(nP)$ number of theorem prover queries, where each query has the VC of at most one procedure only.

For SDV, we generated candidates using two different sources of predicates: first, we used YOGI’s internal heuristics, which compute an initial set of predicates for YOGI’s iterative refinement loop; and second, we manually inspected some properties and wrote down predicates that captured the *typestate* of the property. Because we never looked at the drivers themselves, this process required minimal manual effort. We now briefly illustrate this process.

A driver goes through two instrumentation passes that are important to understand for generating HOUDINI candidates. The first pass instruments the driver with a property. For illustration, consider a property which asserts that for a given lock, acquire and release operations must occur in strict alternation. A driver is instrumented with this property by introducing a new variable s of type `int` that is initialized to 0 in the beginning of `main`. A lock acquire operation first asserts that $s == 0$ and then sets s to 1. A lock release operation first asserts that $s == 1$ and then sets s to 0. It is easy to see that if the instrumented driver does not fail any assertion then the acquire and release operations happen in strict alternation.

The second instrumentation is carried out internally inside CORRAL as a pre-processing step. Because the stratified inlining algorithm only checks for a condition at the end of `main`, CORRAL introduces a new Boolean variable `error`, initialized to `false`, and replaces each assertion `assert e` with:

```
error :=  $\neg e$ ; if(error) return;
```

And after each procedure call, CORRAL inserts:

```
if(error) return;
```

Then CORRAL simply asserts that **error** is *false* at the end of **main**.

HOUDINI candidates are derived from predicates that capture the property typestate. For the lock acquire-release property, the typestate predicates are $s == 0$ and $s == 1$. These are used to generate the following 6 candidates to capture either (a) how the typestate can be modified by a procedure; or (b) conditions under which **error** is not set:

$$\begin{array}{ll} \textit{old}(s) == 0 \Rightarrow s == 0 & \textit{old}(s) == 1 \Rightarrow s == 0 \\ \textit{old}(s) == 0 \Rightarrow s == 1 & \textit{old}(s) == 1 \Rightarrow s == 1 \\ \textit{old}(s) == 0 \Rightarrow \neg \text{error} & \textit{old}(s) == 1 \Rightarrow \neg \text{error} \end{array}$$

Here, *old*(s) refers to the value of s at the beginning of a procedure and s refers to its value at the end of a procedure.

The generated candidates are fed to HOUDINI and valid ones form procedure summaries. These summaries not only help proving correctness of certain programs, but also help in finding bugs faster: when a part of a program is proved correct, no further inlining is required in that region. A scatter plot on the buggy instances is shown in Fig. 8. The running time of HOUDINI is included in the running time of CORRAL. For simple instances, this adds extra overhead: the distribution of points around the origin are in favor of YOGI. However, as running time increases, CORRAL + HOUDINI is almost always faster. On programs with non-trivial running times, CORRAL + HOUDINI was six times faster than YOGI (up from 2X without HOUDINI).

Computing Proofs. CORRAL can prove correctness regardless of the recursion bound when the over-approximation used in stratified inlining is UNSAT. Surprisingly, this simple over-approximation (along with partial inlining) suffices in many cases. Of the 1886 correct programs, CORRAL was actually able to prove 1574 of them correct irrespective of the recursion bound: a proof rate of 83%. With the use of HOUDINI, this number goes up to 1715: a proof rate of 91%.

4.3 SV-COMP Benchmarks

For the next experiment, we looked for external sources of benchmarks that are already accessible to other tools. We found a rich source of such benchmarks from the recently-held competition on software verification [6]. Unfortunately, the benchmarks are CIL-processed C files, most of which do not compile using our (Windows-based) front end. Thus, we picked all programs in only two categories and manually fixed their syntax. The first category consists of 36 programs in the **ssh** folder and the second category consists of 9 programs in the **ntdrivers** folder. Roughly half of these programs are correct, and half are buggy. We ran CORRAL with a recursion bound of 10, which is the same used by other bounded model checkers in the competition. We did not attempt to set up the tools that

participated in the competition on our machine. Instead, we compare CORRAL against the running times reported online. Although it is unfair to compare running times on different machines (and operating systems), we only show the numbers to illustrate CORRAL’s robustness, and a ball-park estimate of how it competes against many tools on a new set of benchmarks.

For each program, CORRAL returned the right answer well within the time limit of 900 seconds. In the `ssh` category, CORRAL takes a total of 168 seconds to finish. This is in comparison to 525.8 seconds taken by the best tool in the category (CPACHECKER). In the `ntdrivers` category, CORRAL took a total of 447.5 seconds, but in this case, the best tool (again CPACHECKER) performed better—it took only 105.2 seconds. No tool other than CORRAL and CPACHECKER was able to return right answers on all programs in `ntdrivers` (they either ran out of time or memory or returned an incorrect answer).

4.4 .NET Benchmarks

We downloaded an open-source .NET implementation of the Tetris game and set it up with a harness that clicks random buttons and menus to drive the game. We compiled this program to BOOGIE using BCT [5]. The original program is around 1550 lines of source code (excluding type definitions and comments); it compiles to roughly 23K lines of BOOGIE code and has 357 procedures. We created a collection of 570 queries, one for each target of each branch in the program as well as the beginning of each procedure. We ran CORRAL with a recursion bound of 1. Within a budget of 600 seconds per query, CORRAL was able to resolve 243 queries as reachable, 204 as unreachable, and the rest (123) timed out. These results, together with results described earlier on C programs, demonstrate CORRAL’s robustness in dealing with queries from different programming languages.

5 Related Work

Stratified inlining is most closely related to previous work on *structural abstraction* [1], *inertial refinement* [3] and *scope bounding* [27][21]. However, structural abstraction is based entirely on overapproximations; it does not have the analog of underapproximating by blocking certain calls. Inertial refinement uses both over and under approximations to iteratively build a view of the program that is a collection of regions and uses the notion of minimal correcting sets [26] for the refinement. The scheme in stratified inlining appears to be much simpler because, first, it abstracts only calls and not arbitrary program regions. Second, the refinement is based on a simple analysis of the counterexample from the over-approximate query. Scope bounding refers to limiting the scope of an analysis to a program fragment and has been used previously, for instance, in verifying null-dereference safety of Java programs [27]. In their context, fragments need not contain the entry procedure and may grow backwards (i.e., callers get inlined,

instead of callees), which is in contrast to stratified inlining. Moreover, the process of growing fragments is not based on abstract counterexamples. Additionally, none of these previous techniques attempt to perform variable abstraction, whereas CORRAL effectively orchestrates it along with stratified inlining.

The idea of introducing Boolean variables for doing optimization inside a theorem prover, which we use in Alg. 2, has been used previously, for instance in error localization [22].

A variety of methods have been proposed for software verification based on predicate abstraction and interpolation [218][29][28]. The focus of these methods is to infer predicates in order to create a finite vocabulary for invariants and summaries. These techniques are complementary to our work; summaries generated by them could be used to speed up CORRAL as shown by our experiments with HOUDINI.

Bounded model checkers such as CBMC [8], ESBMC [11], and LAV [32] perform bounded program verification similar to CORRAL. However, their focus is on efficient VC generation and modeling of program semantics; they use a technique similar to static inlining and do not use variable abstraction.

References

1. Babic, D., Hu, A.J.: Calysto: scalable and precise extended static checking. In: ICSE, pp. 211–220 (2008)
2. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM 54(7), 68–76 (2011)
3. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE, pp. 82–87 (2005)
5. Barnett, M., Qadeer, S.: BCT: A translator from MSIL to Boogie. In: Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (2012)
6. Beyer, D. (ed.): 1st International Competition on Software Verification, co-located with TACAS 2012, Tallinn, Estonia (2012)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
8. Clarke, E.M., Kroning, 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)
9. Clarke, E.M., Kurshan, R.P., Veith, H.: The Localization Reduction and Counterexample-Guided Abstraction Refinement. In: Manna, Z., Peled, D.A. (eds.) Pnueli Festschrift. LNCS, vol. 6200, pp. 61–71. Springer, Heidelberg (2010)
10. Condit, J., Hackett, B., Lahiri, S., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Principles of Programming Languages (2009)
11. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering (2011)
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)

13. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD, pp. 45–52 (2009)
14. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
15. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
16. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Principles of Programming Languages (2011)
17. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
18. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages (2002)
19. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
20. Hopcroft, J.E., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley (1999)
21. Ivancic, F., Balakrishnan, G., Gupta, A., Sankaranarayanan, S., Maeda, N., Tokuoka, H., Imoto, T., Miyazaki, Y.: DC2: A framework for scalable, scope-bounded software verification. In: ASE (2011)
22. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI (2011)
23. Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
24. Lal, A., Qadeer, S., Lahiri, S.: Corral: A solver for reachability modulo theories. Technical Report MSR-TR-2012-09, Microsoft Research (2012)
25. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1) (2009)
26. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning 40(1), 1–33 (2008)
27. Loginov, A., Yahav, E., Chandra, S., Fink, S., Rinetzky, N., Nanda, M.G.: Verifying dereference safety via expanding-scope analysis. In: ISSTA (2008)
28. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
29. Nori, A.V., Rajamani, S.K.: An empirical study of optimizations in YOGI. In: ICSE, pp. 355–364 (2010)
30. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
31. Sinha, N.: Modular bug detection with inertial refinement. In: FMCAD (2010)
32. Vujošević-Janičić, M., Kuncak, V.: Development and Evaluation of LAV: An SMT-Based Error Finding Platform. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 98–113. Springer, Heidelberg (2012)

On Decidability of Prebisimulation for Timed Automata

Shibashis Guha, Chinmay Narayan, and S. Arun-Kumar

Department of Computer Science and Engineering,

Indian Institute of Technology, Delhi

shibashis,chinmay,sak@cse.iitd.ac.in

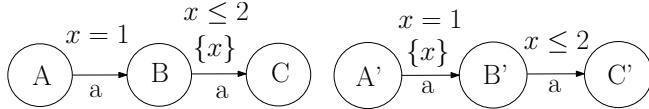
Abstract. In this paper, we propose an *at least as fast as* relation between two timed automata states and investigate its decidability. The proposed relation is a prebisimulation and we show that given two processes with rational clock valuations it is decidable whether such a prebisimulation relation exists between them. Though bisimulation relations have been widely studied with respect to timed systems and timed automata, prebisimulations in timed systems form a much lesser studied area and according to our knowledge, this is the first of the kind where we study the decidability of a timed prebisimulation. This prebisimulation has been termed *timed performance prebisimulation* since it compares the efficiency of two states in terms of their performances in performing actions. $s \precsim t$ if s and t are time abstracted bisimilar and every possible delay by s and its successors is no more than the delays performed by t and its successors where the delays are real numbers. The prebisimilarity defined here falls in between timed and time abstracted bisimilarity.

Keywords: Timed automata, timed bisimulation, time abstracted bisimulation, prebisimulation, timed transition system.

1 Introduction

Bisimulation [17] is an important relation to establish the equivalence between two reactive systems. The concept has been extended to timed systems as well. The common form of bisimulations used to compare two timed systems are *timed* and *time abstracted* bisimulations. While timed bisimulation is too strong a relation where time delays have to match exactly, time abstracted bisimulation does not compare exact timing requirements. Both timed and time-abstracted bisimulations have been proved to be decidable for timed automata [6][16][21].

Timing requirements, in real time systems in particular, increasingly affect design decisions and implementation procedure. We propose a prebisimulation relation between two timed automata states which establishes an *at least as fast as* relation between them. Using action hiding or action abstraction, the proposed bisimulation relation will be useful in comparing functionally equivalent systems in terms of their relative performance. We call this *timed performance prebisimulation* as it distinguishes two states based on their performance for

**Fig. 1.** Example: A preorder relation

doing each action they are capable of doing. This relation will be useful for verifying the correctness of implementations of systems with multiple blocks where each individual block needs to finish execution within a specified time. The verification will conclude whether the timing requirement for each individual block as mentioned in the specification is satisfied in the implementation. An important consideration here is that both the states should be capable of performing the same set of actions at any point in time and hence they should be time abstracted bisimilar. Thus timed performance prebisimulation is stronger than time abstracted bisimulation but weaker than timed bisimulation. The automata shown in figure 1 illustrate the idea. The automaton in the left is *at least as fast as* the automaton on the right, since the second a action should be performed within a time interval of one time unit after the first a action whereas in the second timed automaton, the second a can be performed within an interval of two time units after the first action. Some *speed sensitive* preorder relations have been defined in process algebraic framework [4] [18] [15], though each of these relations is significantly different from our timed performance prebisimulation which is defined for timed automata states. In [12] too a similar relation based on traces is proposed. In our approach we can capture non-determinism in terms of both time and action and the relation captures branching in time as in bisimulation rather than traces. Also, our relation is particular to the widely studied timed automata formalism whereas the formalism used in [12] is timed actor interfaces. As in the work in [12], in our approach too, a smaller nondeterministic delay against a greater constant time delay is considered to be a refinement. Our main contribution is proving the decidability of timed performance prebisimulation using a zone [4] [9] based construction. This approach can also be used to prove the decidability of timed bisimulations. For timed bisimulation, our construction can lead to a simpler proof than the product construction technique used on regions in [6] or the zone based technique used in [21]. In section 2, we give a brief description of timed automata. In section 3, we describe strong timed and time-abstracted bisimulations while in section 4, we introduce *timed performance prebisimulation*. In section 5, we briefly describe zone and zone graph [14] [22]. Our algorithm for proving decidability uses *zone valuation graph* which is a zone graph starting from a specific timed state and satisfies certain properties as described later. We describe a method for creating zone valuation graph in section 5 and provide an algorithm for deciding this prebisimulation and a proof of correctness for it. We conclude in section 6 where we give an example comparing two complex systems using this performance prebisimulation.

We also mention about another timed prebisimulation which we consider to be of significant importance and whose decidability we would like to investigate in future.

2 Timed Automata

Timed automata [2] is an approach to model time critical systems where the system is modeled with *clocks* that track elapsed time. Timing of actions and time invariants on states can be specified using this model.

A timed automaton is a finite-state structure which can manipulate real-valued clock variables. Corresponding to every transition, a subset of the clocks can be specified that can be *reset* to zero. In this paper, the clocks that are reset in a transition are shown as being enclosed in braces. Clock constraints also specify the condition for actions being enabled. If the constraints are not satisfied, the actions will be disabled. Constraints can also be used to specify the amount of time that may be spent in a location. The clock constraints $\mathcal{B}(C)$ over a set of clocks C is given by the following grammar:

$$g ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \wedge g$$

where $c \in \mathbb{N}$ and $x \in C$. A *timed automaton* over a finite set of clocks C and a finite set of actions Act is a quadruple (L, l_0, E, I) [1] where L is a finite set of locations, ranged over by l , $l_0 \in L$ is the initial location, $E \subseteq L \times \mathcal{B}(C) \times Act \times 2^C \times L$ is a finite set of *edges*, and $I : L \rightarrow \mathcal{B}(C)$ assigns invariants to locations.

2.1 Semantics

The semantics of a timed automaton can be described with a *timed labeled transition system*(TLTS). Let $A = (L, l_0, E, I)$ be a timed automaton over a set of clocks C and a set of visible actions Act . The timed transition system $T(A)$ generated by A can be defined as $T(A) = (Proc, Lab, \{\xrightarrow{\alpha} \mid \alpha \in Lab\})$, where $Proc = \{(l, v) \mid (l, v) \in L \times (C \rightarrow \mathbb{R}_{\geq 0}) \text{ and } v \models I(l)\}$, i.e. *states* are of the form (l, v) , where l is a location of the timed automaton and v is a valuation that satisfies the invariant of l . We use the terms *process* and *state* interchangeably in this text. $Lab = Act \cup \mathbb{R}_{\geq 0}$ is the set of labels; and the transition relation is defined by $(l, v) \xrightarrow{a} (l', v')$ if for an edge $(l \xrightarrow{g,a,r} l') \in E$, $v \models g$, $v' = v[r]$ and $v' \models I(l')$, where an edge $(l \xrightarrow{g,a,r} l')$ denotes that l is the source location, g is the guard, a is the action, r is the set of clocks to be reset and l' is the target location. $(l, v) \xrightarrow{d} (l, v + d)$ for all $d \in \mathbb{R}_{\geq 0}$ such that $v \models I(l)$ and $v + d \models I(l)$ where $v + d$ is the valuation in which every clock value is incremented by d . Let v_0 denote the valuation such that $v_0(x) = 0$ for all $x \in C$. If v_0 satisfies the invariant condition of the initial location l_0 , then (l_0, v_0) is the initial state or the initial configuration of $T(A)$.

3 Bisimulations for Timed Systems

We discuss here only the strong form of the bisimulations.

Definition 1. *Timed bisimilarity:* A binary symmetric relation \mathcal{R} over the set of states of a TLTS is a timed bisimulation relation if whenever $s_1 \mathcal{R} s_2$, for each action $a \in \text{Act}$ and time delay $d \in \mathbb{R}_{\geq 0}$

if $s_1 \xrightarrow{a} s'_1$ then there is a transition $s_2 \xrightarrow{a} s'_2$ such that $s'_1 \mathcal{R} s'_2$, and

if $s_1 \xrightarrow{d} s'_1$ then there is a transition $s_2 \xrightarrow{d} s'_2$ such that $s'_1 \mathcal{R} s'_2$.

Timed bisimilarity \sim_t is the largest timed bisimulation relation.

Timed automata A_1 and A_2 are *timed bisimilar* if the initial states in the corresponding TLTS are timed bisimilar. Matching each time delay in one automaton with identical delays in another automaton may be too strict a requirement. Time abstracted bisimilarity is the relation obtained by a relaxation of this requirement and the second clause of definition 1 is replaced by

if $s_1 \xrightarrow{d} s'_1$ then there is a transition $s_2 \xrightarrow{d'} s'_2$, such that $s'_1 \mathcal{R} s'_2$. The delay d can be different from d' .

Timed automata A_1 and A_2 are *time abstracted bisimilar* if the initial states in the corresponding TLTS are time abstracted bisimilar.

4 Timed Performance Prebisimulation

Deciding whether one automaton is at least as fast as another is interesting when they can perform the same visible actions at every stage. Thus they should be time abstracted bisimilar. This prebisimulation is similar to efficiency pre-order 3 where the efficiency of two systems is compared using the number of internal moves made by them.

Definition 2. *Timed performance prebisimilarity:* A binary relation \mathcal{B} over the set of states of a TLTS is a timed performance prebisimulation relation if whenever $s_1 \mathcal{B} s_2$, for each action a and time delay d

if $s_1 \xrightarrow{a} s'_1$ then there is a transition $s_2 \xrightarrow{a} s'_2$ such that $s'_1 \mathcal{B} s'_2$, and

if $s_2 \xrightarrow{a} s'_2$ then there is a transition $s_1 \xrightarrow{a} s'_1$ such that $s'_1 \mathcal{B} s'_2$, and

if $s_1 \xrightarrow{d} s'_1$ then there is a transition $s_2 \xrightarrow{d'} s'_2$ for $d \leq d'$ such that $s'_1 \mathcal{B} s'_2$, and

if $s_2 \xrightarrow{d} s'_2$ then there is a transition $s_1 \xrightarrow{d'} s'_1$ for $d \geq d'$ such that $s'_1 \mathcal{B} s'_2$.

Timed performance prebisimilarity \precsim is the largest timed performance prebisimulation relation.

Clearly $\sim_t \subset \precsim \subset \sim_u$. In this paper, for brevity, we use the term *timed performance prebisimilarity* and *performance prebisimilarity* interchangeably.

5 Zone Valuation Graph and Deciding Prebisimulation

We present a zone graph based algorithm which, given two timed automata states s_1 and s_2 with rational clock valuations, decides whether or not $(s_1, s_2) \in \precsim$.

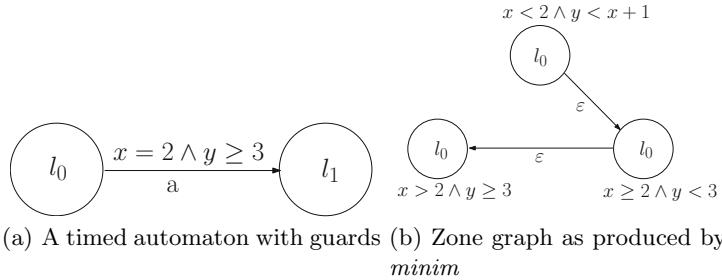


Fig. 2. A timed automaton in (a) and its zone valuation graph corresponding to process $\langle l_1, (0, 0.6) \rangle$ in (b)

We first discuss the definition of a zone graph and the properties that must hold for it to be applicable in our algorithm. A zone graph that satisfies these properties is referred to as a *zone valuation graph*. The concept is introduced in subsection 5.1 and is used in subsection 5.2 to prove that the prebisimulation relation is decidable. In subsection 5.3, we give an algorithm for constructing *zone valuation graphs* and discuss its complexity. After this we present an algorithm in subsection 5.4 which uses this *zone valuation graph* to check the existence of a performance prebisimulation relation between two states of timed automata.

5.1 Zone Valuation Graph

The following two definitions are from [21].

Definition 3. zone: The characteristic set of a linear formula ϕ , a clock constraint of the form $x \sim c$ or a diagonal constraint of the form $x - y \sim c$, where $x, y \in C$, is the set of all valuations for which ϕ holds. A zone is a finite union of characteristic sets.

A zone graph is similar to a region graph [1] with the difference that each node consists of a timed automaton location and a zone.

Definition 4. zone graph: For a timed automaton $P = (L, l_0, E, I)$, a zone graph is a transition system $(S, s_0, Lep, \rightarrow)$, where $Lep = Act \cup \{\varepsilon\}$, ε is an action corresponding to delay transitions of the processes of the zone, $S \subseteq L \times \Phi_V(C)$ is the set of nodes, $s_0 = (l_0, \phi_0(C))$, $\rightarrow \subseteq S \times Lep \times S$ is connected, $\phi_0(C)$ is the formula where all the clocks in C are 0 and $\Phi_V(C)$ denotes the set of all zones.

Definition 5. Bisimulation between zone graphs

For two zone graphs, $Z_1 = (S_1, s_1, Lep, \rightarrow_1)$ and $Z_2 = (S_2, s_2, Lep, \rightarrow_2)$, Z_1 is strongly bisimilar [17] to Z_2 , denoted as $Z_1 \sim Z_2$, iff the nodes s_1 and s_2 are strongly bisimilar, denoted by $s_1 \sim s_2$. While checking strong bisimulation between the two zone graphs, ε is considered visible similar to an action in Act .

It is possible to have more than one zone graph corresponding to a timed automaton. For a timed automaton $A = (L, l_0, E, I)$ and a process $r = (l_j, v_{l_j}) \in T(A)$,

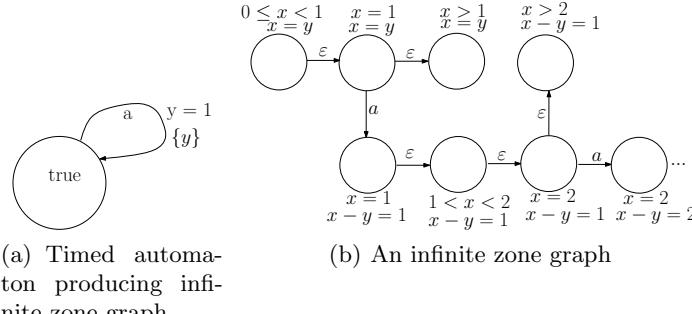


Fig. 3. An automaton in (a) with its infinite zone graph in (b)

we are interested in a particular form of zone graph $Z_{(A,r)} = (S, s_r, Lep, \rightarrow)$ which satisfies the following properties:

1. set S is finite.
2. For every node $s \in S$ the zone corresponding to the constraints ϕ_s is convex.
3. $v_{l_j} \models \phi_{s_r}$. Note that v_{l_j} may or may not satisfy $\phi_0(C)$.
4. For any two processes $p, q \in T(A)$, if their valuation satisfies the formula ϕ_r for the same node $r \in S$ then $p \sim_u q$, i.e. p is time abstracted bisimilar to q .
5. For two timed automata A_1, A_2 and two processes $p \in T(A_1)$ and $q \in T(A_2)$, $Z_{(A_1,p)} \sim Z_{(A_2,q)} \Leftrightarrow p \sim_u q$.
6. It should be minimal to the extent of preserving convexity of the zones and gives a canonical form for checking performance prebisimulation.

Zone graph created with the algorithm given in section 5.3 of [20] satisfies most of these properties. It is finite by construction and does not require any abstraction mechanism to ensure finiteness of the zone graph. All the zones are convex and any two processes satisfying the same zone formula in the same node are time abstracted bisimilar. The algorithm described there has been implemented in the tool *minim*. Though for most of the timed automata the zone graph constructed by *minim* preserves all the properties mentioned above, we found that it does not produce the minimal graph preserving time abstracted bisimulations for certain timed automata where all locations are not reachable. This is due to the fact that in *minim* the zones are split based on canonical decomposition of the guards on the outgoing edges. For example, let us look at the automaton in figure 2(a). Due to the guards on the outgoing edges the initial zone of location l_0 is split into sub-zones by *minim* as shown in figure 2(b). However, no valuation in l_0 can satisfy the guard and therefore cannot make an a transition. Therefore a zone graph algorithm producing the minimal graph should not split the zone.

While *minim* uses a backward method, in our approach, while generating the zone valuation graph, we use a combination of both forward and backward analysis. One must note that forward analysis may cause a zone graph to become infinite [10]. Let us consider the automaton in figure 3(a). Since clock y is reset while x keeps increasing, the number of zones becomes infinite as shown in figure

3(b). To ensure finiteness of the zone graph, we require an abstraction on the zones. Several zone abstractions have been suggested in the literature [8] [10] [11]. We consider location dependent maximal constants abstraction [10]. For each clock $x \in C$ and each location $l \in L$, a maximum constant \max_x^l is determined beyond which the actual value of x in l is irrelevant. For a location l and a clock x , $\max_x^l \leq c_x$, the global maximal constant with which clock x is compared. This reduces the number of zones compared to the one obtained using region graph abstraction.

Our algorithm constructs a zone graph whose nodes are the time abstracted bisimulation classes of the automaton preserving the convexity of zones. Thus it is possible to have two or more nodes in the zone graph which are time abstracted bisimilar to each other. Similar to *minim* [20], we use the guards on transitions to split a zone corresponding to a location but unlike *minim*, it is split only when it is found to be reachable from the initial location using a forward analysis [10] [5] approach. One location can be reached through multiple paths and thus a split of a zone in a location also causes its *ancestors* to be split accordingly. This is done using the backward method as in *minim*. Like *minim*, the notion of *stability* is used for the purpose of splitting a zone. We present below a brief description of stability as given in [20]. Given an edge $v \rightarrow u$, where v and u are zones in a zone graph, v is a *predecessor* of u and u is a successor of v where u and v are nodes in a zone graph. Function $\text{preds}(v)$ (resp. $\text{succs}(v)$) denotes the set of predecessors (resp. successors) of v .

Definition 6. Stable partitions: Given two classes $C_1, C_2 \in \mathcal{C}$, C_1 is said to be pre-stable with respect to C_2 if either $C_1 \subseteq \text{preds}(C_2)$ or $C_1 \cap \text{preds}(C_2) = \emptyset$. C_2 is said to be post-stable with respect to C_1 if either $C_2 \subseteq \text{succs}(C_1)$ or $\text{succs}(C_1) \cap C_2 = \emptyset$. In this paper, we use the term *stability* which implies either pre-stability or post-stability depending on the particular situation. We say that C_1 and C_2 are stable with respect to each other if C_1 is pre-stable with respect to C_2 and C_2 is post-stable with respect to C_1 .

Figure 4(a) shows the zone graph after using abstraction with location dependent maximal constants for the timed automaton of figure 3(a). The nodes in the polygonal enclosures are strongly bisimilar to each other such that their union gives a convex zone. Such nodes in the enclosures are combined in the next phase to obtain the *canonical* form of the zone graph shown in figure 4(b). The nodes in the canonical zone graph represent the time abstracted bisimilar classes of the zones preserving convexity.

5.2 Decidability of Timed Performance Prebisimulation

We now discuss how zone valuation graph can be used in checking the existence of timed performance prebisimulation relation between two TLTS states. For this purpose, we first introduce a few notations and definitions which will be used in proving the correctness of our approach.

For timed automaton A and process $p \in T(A)$, let $Z_{(A,p)} = (S, s_0, Lep, \rightarrow)$ be the zone valuation graph of p . For any state $q \in T(A)$, let $\mathcal{N}(q)$ represent the

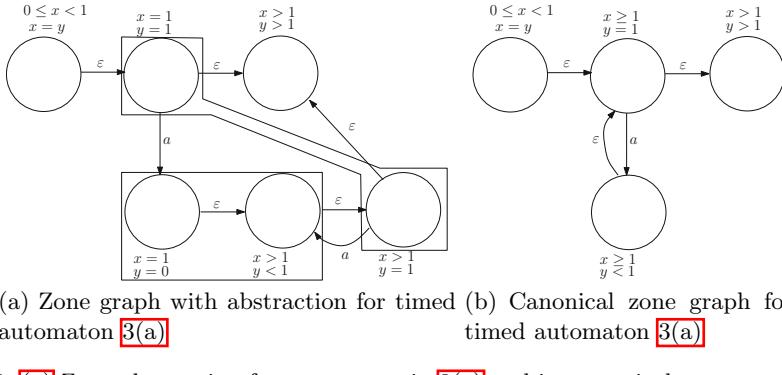


Fig. 4. (a) Zone abstraction for automaton in 3(a) and its canonical representation in (b)

node of the same location as that of q and whose clock valuation range includes the valuation of q . Similarly, for any node $s \in S$, let $\mathcal{G}(s)$ represent the set of all processes reachable from p with the same location as that of s and whose valuations are satisfied by ϕ_s .

Based on the clock constraints of the nodes in the *zone valuation graph*, we define **span** as below,

Definition 7. Span: For a given node $s \in S$ and a clock $x \in C$, $\min_x(s)$ and $\max_x(s)$ represent the minimum and the maximum clock valuations of a clock x across all processes in the node s . For $x \geq c$, $\min_x(s) = c$, for $x > c$, $\min_x(s) = c + \delta$. For $x \leq c$, $\max_x(s) = c$, for $x < c$, $\max_x(s) = c - \delta$, where δ is a symbolic value representing an infinitesimally small value. We define $\text{range}(x, s)$ as $\max_x(s) - \min_x(s)$. The span of a node $s \in S$ is defined as $\mathcal{M}(s) = \min\{\text{range}(x, s) \mid x \in C\}$, i.e. minimum of all clocks' ranges. We define a clock y belonging to the set $\{y \mid \text{range}(y, s) = \mathcal{M}(s)\}$ to be a critical clock.

It is to be noted that for a given node s , $\text{range}(x, s)$ is the same for all clock variables $x \in C$ if the zone corresponding to node s is not abstracted with respect to any clock variable. If the zone corresponding to s is abstracted with respect to one or more clock variables then for each such variable $x \in C$, $\text{range}(x, s) = \infty$. For example, in a zone valuation graph with two clocks x and y , the span for a node s with $\phi_s = x > 3$ and $y < 1$ is $\min(\infty, 1 - \delta) = 1 - \delta$ whereas span for a node with $\phi_s = x > 1$ and $y = 2$ is $\min(\infty, 0) = 0$.

Definition 8. Flip in delay: For timed automata A_1 and A_2 and processes $p \in T(A_1)$, $q \in T(A_2)$, let $Z_{(A_1, p)} = (S_1, s_1, Lep, \rightarrow_1)$ and $Z_{(A_2, q)} = (S_2, s_2, Lep, \rightarrow_2)$ be the corresponding zone valuation graphs. If $Z_{(A_1, p)} \sim Z_{(A_2, q)}$ then flip in delay, denoted by $FID(Z_{(A_1, p)}, Z_{(A_2, q)})$ is true iff there does not exist any strong bisimulation relation \mathcal{B} between $Z_{(A_1, p)}$ and $Z_{(A_2, q)}$, such that the following hold,

- $\forall(s, t) \in \mathcal{B} : s \in S_1, t \in S_2, \mathcal{M}(s) \leq \mathcal{M}(t)$ or
- $\forall(s, t) \in \mathcal{B} : s \in S_1, t \in S_2, \mathcal{M}(s) \geq \mathcal{M}(t)$

i.e. in none of the bisimulation relations, all bisimilar nodes maintain the same relation (less or greater or equal) between their spans.

Definition 9. Given a timed automaton A , let $Z_{(A,p)}$ be the zone valuation graph corresponding to process $p \in T(A)$. Let $p' \in T(A)$ be a process reachable from p and s be the node of $Z_{(A,p)}$ such that $p' \in \mathcal{G}(s)$. Let x be a critical clock of s and $v_{p'}(x)$ denote the valuation of clock x for process p' . We define maximum admissible delay for p' in s as $\max_x(s) - v_{p'}(x)$.

For example, for the process $\langle l_0, 3 \rangle$ in $\textcolor{red}{\square}$ the maximum admissible delay is $5 - 3 = 2$.

Now we are ready to prove the decidability of checking prebisimilarity. Intuitively, the approach consists of checking the strong bisimilarity between two zone valuation graphs and simultaneously checking for the flip in delay between bisimilar nodes. If zone valuation graphs are strongly bisimilar and no flip in delay is found then the processes are prebisimilar, otherwise they are not prebisimilar. The following lemmas prove this intuition.

Lemma 1. For $p \in T(A_1)$ and $q \in T(A_2)$, $FID(Z_{(A_1,p)}, Z_{(A_2,q)}) \wedge p \sim_u q \Rightarrow (p \not\precsim q \wedge q \not\precsim p)$

Proof. $p \sim_u q$ and $FID(Z_{(A_1,p)}, Z_{(A_2,q)})$ imply that in each strong bisimulation relation $\mathcal{B} \subseteq S_1 \times S_2$, such that $(s_{p1}, s_{q1}) \in \mathcal{B}$ and $(s_{p2}, s_{q2}) \in \mathcal{B}$, where $s_{p1}, s_{p2} \in Z_{(A_1,p)}$ and $s_{q1}, s_{q2} \in Z_{(A_2,q)}$, either

- $\mathcal{M}(s_{p1}) > \mathcal{M}(s_{q1})$ and $\mathcal{M}(s_{p2}) < \mathcal{M}(s_{q2})$, or
- $\mathcal{M}(s_{p1}) < \mathcal{M}(s_{q1})$ and $\mathcal{M}(s_{p2}) > \mathcal{M}(s_{q2})$.

We first show that $p \sim_u q$ and $\mathcal{M}(s_{p1}) > \mathcal{M}(s_{q1}) \Rightarrow p \not\precsim q$. Consider the process $p_1 \in \mathcal{G}(s_{p1})$ reachable from p such that $v_{p_1}(x) = \min_x(s_{p1})$, where x is a critical clock of s_{p1} . Consider the delay $p_1 \xrightarrow{\mathcal{M}(s_{p1})} p'_1$. So $p'_1 \in \mathcal{G}(s_{p1})$. There does not exist a $q_1 \in \mathcal{G}(s_{q1})$ such that $q_1 \xrightarrow{d} q'_1$ where $d > \mathcal{M}(s_{p1})$ and $q'_1 \in \mathcal{G}(s_{q1})$. This implies that there does not exist a process q_1 such that $p_1 \precsim q_1$ implying $p \not\precsim q$. Similarly, we can show that $\mathcal{M}(s_{p2}) < \mathcal{M}(s_{q2})$ implies $q \not\precsim p$.

For the second case too we can similarly show that neither $p \precsim q$ nor $q \precsim p$ holds. \square

Lemma 2. For $p \in T(A_1)$ and $q \in T(A_2)$, $p \sim_u q \wedge \neg FID(Z_{(A_1,p)}, Z_{(A_2,q)}) \Rightarrow p \precsim q \vee q \precsim p$.

Proof. Without loss of generality, say in a strong bisimulation relation $\mathcal{B} \subseteq S_1 \times S_2$, span of all nodes in $Z_{(A_2,q)}$ is less than the span of their corresponding bisimilar nodes in $Z_{(A_1,p)}$. We show that this implies $q \precsim p$. Consider some node s_{q1} of $Z_{(A_2,q)}$. Let s_{p1} be a node of $Z_{(A_1,p)}$ such that $s_{p1} \sim s_{q1}$ and q_1 be any process in $\mathcal{G}(s_{q1})$ reachable from q . We will show that there exists a process $p_1 \in \mathcal{G}(s_{p1})$ reachable from p such that $q_1 \precsim p_1$.

Let $d_1 = v_{q_1}(x) - \min_x(s_{q_1})$, where x is a critical clock of s_{q_1} and $d_2 = d_1 \times (\mathcal{M}(s_{p_1})/\mathcal{M}(s_{q_1}))$. If y is a critical clock of s_{p_1} , then we define $p_1 \in \mathcal{G}(s_{p_1})$ to be a process such that $v_{p_1}(y) = \min_y(s_{p_1}) + d_2$. It is easy to see that there exists such a p_1 reachable from p and $q_1 \precsim p_1$.

Now consider a node s_{p_2} in $Z_{(A_1,p)}$. Let s_{q_2} be a node in $Z_{(A_2,q)}$ reachable from q such that $s_{q_2} \sim s_{p_2}$ and let $p_2 \in \mathcal{G}(s_{p_2})$ be a process reachable from p . Let $d_3 = v_{p_2}(z) - \min_z(s_{p_2})$, where z is a critical clock of s_{p_2} and $d_4 = d_3 \times (\mathcal{M}(s_{q_2})/\mathcal{M}(s_{p_2}))$. If w be a critical clock of s_{q_2} , then we define $q_2 \in \mathcal{G}(s_{q_2})$ to be a process such that $v_{q_2}(w) = \min_w(s_{q_2}) + d_4$. It is easy to see that there exists such a q_2 reachable from q such that $q_2 \precsim p_2$.

So we have proved that if there is a strong bisimulation relation in which the span of each node of $Z_{(A_2,q)}$ is less than the span of its corresponding bisimilar node of $Z_{(A_1,p)}$, then $q \precsim p$. Similarly, we can also show that if there exists a strong bisimulation relation in which if the span of each node of $Z_{(A_1,p)}$ is smaller than the span of its corresponding bisimilar node in $Z_{(A_2,q)}$ then $p \precsim q$. Thus for $p \in T(A_1)$ and $q \in T(A_2)$, $p \sim_u q \wedge \neg FID(Z_{(A_1,p)}, Z_{(A_2,q)}) \Rightarrow p \precsim q \vee q \precsim p$. \square

Corollary 1. For $p \in T(A_1)$ and $q \in T(A_2)$, $q \precsim p$ or $p \precsim q \Rightarrow p \sim_u q$ and $\neg FID(Z_{(A_1,p)}, Z_{(A_2,q)})$

Proof. Immediate from the contrapositive of lemma \square \square

Theorem 1. For $p \in T(A_1)$ and $q \in T(A_2)$, $q \precsim p$ or $p \precsim q \Leftrightarrow p \sim_u q$ and $\neg FID(Z_{(A_1,p)}, Z_{(A_2,q)})$

Proof. From lemma \square and corollary \square \square

5.3 Generating Zone Valuation Graph

Algorithm \square describes the construction of the zone valuation graph for a timed automaton process. It uses a combination of both forward and backward traversal of the timed automata locations to create stable partitions of the zones. Initially the timed automaton is traversed so as to find \max_x^l for each clock $x \in C$ and location $l \in L$. This step is used for abstraction purposes. The algorithm then splits a convex polyhedron \square defined by all clocks with valuations $\mathbb{R}_{\geq 0}$ into multiple convex polyhedra according to the clock constraints in the timed automaton. To achieve this, the algorithm traverses the timed automaton in a breadth-first manner using the queue Q . Corresponding to each location l of the timed automaton, a height balanced sorted tree T_l is maintained that stores the locations whose zone splitting effect, based on a certain canonical decomposition of clock constraints as described below, have already been propagated to l . In other words, zones of l are stable with respect to the outgoing transitions from the locations present in T_l . Stability checking considers zones of those locations which are either in *preds* or in *succs* relation with zones in l . These relations are dynamically updated as edges are inserted between the nodes of the zone valuation graph. In the algorithm, l_p denotes the latest location added to T_l and

the zones of l are split so as to make them *stable* with respect to the zones of l_p . A reachable location l is added to Q only if its zones are stable with respect to zones of all locations in T_{l_p} . Function $\text{elements}(L_l)$ returns the set of all elements present in L_l .

For each location l dequeued from Q , its zones are split based on a canonical decomposition of certain clock valuations. The canonical decomposition is obtained from the guards on the outgoing edges of l , the invariant on l and the invariant on the destination location corresponding to each outgoing transition. The splitting of zones of l is further propagated to zones of other locations in T_l . Each $l_j \in T_l$ is also added to the queue since the split of the zones of l_j can cause splits of zones of its descendants as well. The abstraction, if necessary, is done simultaneously during the creation of the zone. A location l_i which is a descendant or ancestor of location l is not added to the queue if its zones are stable with respect to the zones of all locations in T_l . This phase of the algorithm terminates when the queue becomes empty. While constructing zone valuation graph, after phase 1, we are left with a set of nodes in the zone valuation graph. The aim of the next phase is to identify the nodes that are strongly bisimilar to each other and then to combine them. Given a finite graph (labeled transition system), the Paige-Tarjan algorithm [19] produces the largest bisimulation relation. The nodes that are identified to be bisimilar are merged to produce a single node as long as the merged node preserves convexity. In the algorithm, to avoid clutter, we have not specifically mentioned how to add edges. Figure 5(e) shows the zone valuation graph for the state $\langle l_0, 0 \rangle$ corresponding to the timed automaton shown in figure 5(a). All loops in the zone valuation graph have an implicit self loop labeled with ε .

Complexity: In the worst case, corresponding to the timed automaton for which region equivalence matches time abstracted bisimulation, the zone graph created is the same as the region graph. Hence the worst case complexity for zone graph creation is exponential in the number of clocks. We consider the complexity of zone graph creation in terms of a specific input automaton to be of particular interest and thus present the complexity in terms of the number of locations, transitions and clock variables of the automaton and the size of zone valuation graph created after abstraction.

For the purpose of abstraction, a preprocessing step is required to identify \max_x^l for each clock $x \in C$ and each location $l \in L$. From [10], the complexity is $O(t^3)$ where $t = |C| \times n$ and n is the number of locations in the timed automaton. Since each node can be added to Q a maximum of n times, the total number of additions to and deletions from Q is bounded by $O(n^2)$. Let $|S|$ and m denote the number of zones and edges respectively in the zone valuation graph produced after abstraction. For each dequeue operation we list the sub-operations below and find their complexity. Let $l = \text{dequeue}(Q)$.

- For subsequent splits of zones of l , it is required to find the zones of the locations in T_l with respect to which, the zones of l are unstable. This can be done in $O(|E|)$ time.

Algorithm 1. Construction of Zone Valuation Graph*Input: Process $p \in T(A)$ and description of a timed automaton A* *Output: Zone valuation graph corresponding to p*

```

1: Calculate  $\max_x^l$  for each location  $l \in L$  and each clock  $x \in C$ . This is required for
   abstraction to ensure finite number of zones in the zone graph.
2: Initialize  $Q$  to an empty queue. For each location  $l$ , create a zone  $\mathcal{N}$  with clock
   valuations for each clock  $\mathbb{R}_{\geq 0}$ .
3: Associate a height balanced tree  $T_l$  with each location  $l$  of  $A$  and initialize  $T_l$  to be
   empty.
4: Let  $l_0$  be the initial location of  $A$ .
5: Enqueue( $Q, l_0$ ).
6: while  $Q$  not empty do
7:    $l = \text{dequeue}(Q)$ 
8:   if  $T_l$  is not empty then,
9:     Let  $l_p$  be the latest location added to  $T_l$ . /* parent of  $l$  in current path */
10:    Split current zones of  $l$  in order to make them stable with respect to the
      zones of  $l_p$ .                                         /* Forward method */
11:    Abstract each of the newly created zones if necessary.
12:   end if
13:   if  $l \notin T_l$  then
14:     Find the canonical decomposition of constraint based on the guards of the
      outgoing transitions of  $l$  along with the invariant of  $l$  and invariant of destination
      location of each transition.
15:     Split  $l$  further based on this canonical decomposition mentioned above.
16:     Abstract each of the newly created zones if necessary.
17:     Add  $l$  to  $T_l$ 
18:   end if
19:   for all location  $l_j \in T_l$  do
20:     If zones of  $l_j$  are unstable with respect to zones of  $l$ , then split zones of  $l_j$ 
      to make them stable with respect to zones of  $l$ .
21:     Abstract each of the newly created zones if necessary.
22:     if  $l \notin T_{l_j}$  then Add  $l$  to  $T_{l_j}$ 
23:     end if
24:     if  $\text{elements}(T_{l_j}) \setminus \text{elements}(T_l) \neq \emptyset$  then
25:       Enqueue( $Q, l_j$ ) /* An ancestor is enqueued: backward method */
26:     end if
27:   end for
28:   for all successor  $l_i$  of  $l$  do /* Successor in timed automaton */
29:     if  $l_i$  is reachable from any of the current set of zones of  $l$  then
30:       if  $l \notin T_{l_i}$  then Add  $l$  to  $T_{l_i}$ 
31:       end if
32:       if  $\text{elements}(T_{l_i}) \setminus \text{elements}(T_l) \neq \emptyset$  then
33:         Enqueue( $Q, l_i$ ) /* A successor is enqueued */
34:       end if
35:     end if
36:   end for
37: end while
38: Obtain canonical form of the zone valuation graph by merging time abstracted
   bisimilar nodes of the zone graph while preserving convexity. /* Phase 2 */

```

- For all locations in T_l , the number of zones that are unstable with respect to the zones of l and hence will be split are of order $O(|S|)$. Hence finding such zones, which will be split due to split of l based on the guards on outgoing edges of l , requires $O(|S| \times |C|)$ time, where the multiplier $|C|$ arises from the fact that each zone is defined by $|C|$ clocks. Abstraction of each zone also requires the same complexity.
- The complexity of split of l based on canonical decomposition is $O(|S| \times |C|)$.
- The number of locations that can be reached using the outgoing edges from l is of order $O(n)$. Before each such location l_i is enqueued to Q , its stability is checked with respect to the zones of all locations appearing in T_l which requires a set difference operation between $\text{elements}(T_{l_i})$ and $\text{elements}(T_l)$ that can be done in $O(n \log n)$. If the corresponding zones are already stable with respect to zones belonging to all locations in T_l , then it is not added to Q . This condition along with the abstraction ensures termination of the zone graph construction procedure. As the number of locations in T_l is $O(n)$, for $O(n)$ operations this step has complexity of order $O(n^2 \times \log n)$.

Corresponding to every dequeue operation, the cost incurred is $O(|E| + |S| \times |C| + n^2 \times \log n) = O(|S| \times |C| + n^2 \times \log n)$. Thus considering $O(n^2)$ dequeue operations, the total computation cost of phase 1 including the preprocessing phase to identify \max_x^l for each clock $x \in C$ is $O(|S| \times |C| \times n^2 + n^4 \times \log n)$.

Phase 2: Here we compute the complexity of the phase where the nodes in the zone graph obtained after phase 1 that are time abstracted bisimilar to each other are merged to produce the minimal zone graph preserving convexity of zones. We use Paige-Tarjan [19] partition refinement algorithm which has a cost of $O(|S| \times \log m)$. This gives us the set of nodes that are bisimilar to each other and hence can be merged. The initial partition is decided in a way so as to ensure that nodes having non-convex zones are not merged.

Thus the total cost of computation for construction of the zone valuation graph is $O(n^2(|C|^3n + |S| \times |C| + n^2 \times \log n) + |S| \times \log m)$.

An Example of Generating the Zone Valuation Graph through Stages: A timed automaton is shown in part (a) of figure 5. Parts (b) through (e) show steps in the generation of the corresponding zone valuation graph. The queue Q is initialized with location l_0 , the initial location of the timed automaton. The zone valuation of location l_0 is split according to the canonical decomposition of the outgoing clock conditions of location l_0 . After this split, T_{l_0} is initialized to $\{l_0\}$ and l_1 is enqueued to Q . Following a dequeue operation, l_1 zones are split to make it stable with respect to the zones of l_0 . l_1 is also split according to the canonical decomposition of its outgoing transition. The effect of this split is again propagated to l_0 backwards which too is accordingly split. The process continues until the zones of the locations cannot be further split. We note here that the split happens in both forward and backward direction. The sequence of enqueue and dequeue operations for this particular example according to algorithm 1 are as follows:

$\text{enqueue}(l_0), \text{dequeue}(l_0), \text{enqueue}(l_1), \text{dequeue}(l_1), \text{enqueue}(l_0), \text{enqueue}(l_2), \text{dequeue}(l_0), \text{dequeue}(l_2), \text{enqueue}(l_1), \text{enqueue}(l_0), \text{dequeue}(l_1), \text{dequeue}(l_0)$.

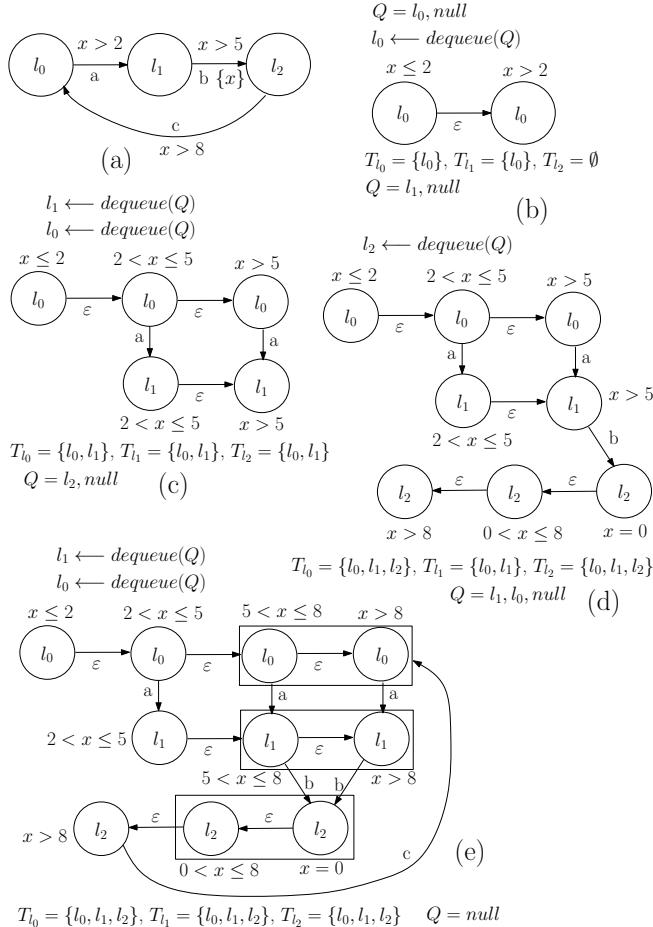


Fig. 5. Timed automaton in part (a) and successive steps in creation of its zone valuation graph

The diagram shows the dequeue operations above each of the figures. Below each figure, the state of the queue is described after the operation shown in the figure gets completed. In part (e), the boxes enclose the strongly bisimilar zones that are combined in phase 2 to create the canonical zone valuation graph.

5.4 Algorithm: Deciding Timed Performance Prebisimulation Using Zone Valuation Graph

In algorithm 2, we present a method to decide if two states $p \in T(A)$ and $q \in T(B)$ are timed performance prebisimilar. After creating the zone valuation graphs for p and q , it invokes `checkTimedPrebisim` method with the initial nodes s_p and s_q of these zone valuation graphs and the binary comparison

Algorithm 2. Algorithm for deciding timed performance prebisimulation

Inputs: i) Two timed automata states p and q ii) description of the timed automata.

Output: Decision whether p and q are timed performance prebisimilar.

```

1: Create zone valuation graphs for  $p$  and  $q$ 
2: Let  $s_p$  and  $s_q$  be the initial nodes in these zone valuation graphs
3: if  $\mathcal{M}(s_p) \leq \mathcal{M}(s_q)$  and  $\text{CHECKTIMEDPREBISIM}(s_p, s_q, \leq)$  then
4:   Declare  $s_p \lesssim s_q$ 
5: else
6:   if  $\mathcal{M}(s_p) \geq \mathcal{M}(s_q)$  and  $\text{CHECKTIMEDPREBISIM}(s_p, s_q, \geq)$  then
7:     Declare  $s_q \lesssim s_p$ 
8:   else
9:     Declare  $p$  and  $q$  are not timed performance prebisimilar
10:  end if
11: end if

12: procedure  $\text{CHECKTIMEDPREBISIM}(s_p, s_q, \text{Rel})$ 
13:   if  $(s_p, s_q) \in L_{pbisim}$  then return true
14:   end if
15:   if  $(s_p, s_q) \in L_{notpbisim}$  then return false
16:   end if
17:   A := sort( $s_p$ ); B := sort( $s_q$ );
18:   if  $A \neq B$  then Add  $(s_p, s_q)$  to  $L_{notpbisim}$ ; return false
19:   end if
20:   Add  $(s_p, s_q)$  to  $L_{pbisim}$ 
21:   for all successors  $s'_p$  of  $s_p$  do
22:     if  $s_p \xrightarrow{\alpha} s'_p$  then                                /* Here  $\alpha \in Act \cup \{\varepsilon\}$  */
23:       r := false
24:       for all  $s'_q$  such that  $s_q \xrightarrow{\alpha} s'_q$  and  $(\mathcal{M}(p') \text{ Rel } \mathcal{M}(q'))$  do
25:         r :=  $\text{CHECKTIMEDPREBISIM}(s'_p, s'_q, \text{Rel})$ 
26:         If r then break; else continue EndIf
27:       end for
28:       if not(r) then
29:         Remove  $(s_p, s_q)$  from  $L_{pbisim}$ ; Add  $(s_p, s_q)$  to  $L_{notpbisim}$ ; return false
30:       else continue
31:     end if
32:   end if
33: end for
34: for all successors  $s'_q$  of  $s_q$  do
35:   if  $s_q \xrightarrow{\alpha} s'_q$  then
36:     r := false
37:     for all  $s'_p$  such that  $s_p \xrightarrow{\alpha} s'_p$  and  $(\mathcal{M}(p') \text{ Rel } \mathcal{M}(q'))$  do
38:       r :=  $r \vee \text{CHECKTIMEDPREBISIM}(s'_p, s'_q, \text{Rel})$ 
39:       If r then break; else continue EndIf
40:     end for
41:     if not(r) then
42:       Remove  $(s_p, s_q)$  from  $L_{pbisim}$ ; Add  $(s_p, s_q)$  to  $L_{notpbisim}$ ; return false
43:     else continue
44:   end if
45: end if
46: end for
47: return true
48: end procedure

```

operator Rel . Passing \leq as Rel is equivalent to checking if $p \precsim q$ and similarly passing \geq is equivalent to checking if $q \precsim p$. If both calls return false then the states p and q are declared not timed performance prebisimilar. If either of them returns true then the states p and q are declared timed performance prebisimilar. It can be shown that by passing $=$ operator as Rel , we can check timed bisimilarity between p and q [13]. The function `checkTimedPrebisim` checks strong bisimilarity between two nodes whose spans are in the relations Rel with each other. Function `sort(s_p)` returns the set of actions that can be performed by the node s_p . Two search data structures L_{pbsim} and $L_{notpbsim}$ are maintained by this function. All pairs of nodes which have been proved not to be prebisimilar are kept in $L_{notpbsim}$. L_{pbsim} maintains all those pair of nodes that have been encountered so far and either have been assumed to be prebisimilar or have been proved to be prebisimilar. These data structures are used to avoid checking the same pair of nodes repeatedly for prebisimilarity. They also make sure that the function `checkTimedPrebisim` is invoked at most $n_1 n_2$ times where n_1 and n_2 represent the number of nodes in the zone valuation graphs of p and q respectively. Complexity of one such invocation, because of two nested for loops, is $O(m_1 m_2 |C|)$ where m_1 and m_2 represent the number of edges in the zone valuation graphs of p and q respectively. The factor for C appears because of the complexity of checking the span. Similarly the complexity of searching, inserting and removing the elements from search data structures L_{pbsim} and $L_{notpbsim}$ is bounded by $O(n_1 n_2 \log(n_1 n_2))$. Therefore the total complexity of this algorithm is $O(n_1 n_2 m_1 m_2 |C|) \times O(n_1 n_2 \log(n_1 n_2)) = O(n_1^2 n_2^2 \cdot m_1 m_2 |C| \log(n_1 n_2))$.

Theorem 2. *Algorithm 2 determines whether a timed performance prebisimulation relation exists between two processes p and q , i.e. if $p \precsim q$ or $q \precsim p$.*

Proof. The algorithm declares two processes to be performance prebisimilar if there exists a strong bisimulation relation between their zone valuation graphs with no flip in delay otherwise the algorithm declares them as not performance prebisimilar. The correctness of the algorithm thus follows from theorem 1. \square

6 Conclusion and Future Work

In the technical report [13], we show how timed performance prebisimulation can be used for showing that one system is at least as fast as another. We consider two protocols for reliable data transfer, alternating bit protocol and Stop-and-Wait ARQ and show that alternating bit protocol is ‘at least as fast as’ stop-and-wait ARQ. Each protocol model consists of a sender, receiver and a lossy channel. Each of sender, channel and receiver is modeled with timed automata and both the systems consist of the parallel composition of the three. Using this prebisimulation relation, we formally prove that Alternating bit protocol is at least as fast as Stop-and-Wait ARQ. We cannot provide the details of the examples due to space constraint. The interested reader is referred to [13] for details of modeling of the two systems using timed automata.

Though bisimulation is a standard notion of program equivalence, prebisimulations have not been studied much in the literature on timed automata. However we do believe that “faster than” preorders are important in real time systems where “responsiveness” is an important aspect of the specification and in the implementation. In this paper, we have defined timed performance prebisimulation and proved its decidability using zone valuation graph for two timed automata states. We also show the use of *zone valuation graph* for checking timed bisimilarity between two processes. In future, we plan to define a weaker prebisimulation relation in which one state can be defined to be at least as fast as the other state if the time elapsed is compared over sequence of actions instead of comparing delays at every stage as in timed performance prebisimulation. We also look forward to investigate the closure properties of these relations.

Acknowledgements. We would like to thank the anonymous referees for their constructive comments and suggestions. We also thank Stavros Tripakis for pointing us to many of the state of the art works that are related to our contribution. This research was partly sponsored by EADS and partly by Microsoft Research India Travel Grants.

References

1. Aceto, L., Ingólfssdóttir, A., Larsen, K.J., Srba, J.: *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press (2007)
2. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* 126, 183–235 (1994)
3. Arun-Kumar, S., Hennessy, M.: An efficiency preorder for processes. *Acta Informatica* (1992)
4. Bengtsson, J.E., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Lectures on Concurrency and Petri Nets*. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
5. Bouyer, P.: Forward analysis of updatable timed automata. *Formal Methods System Design* 24(3), 281–320 (2004)
6. Cerans, K.: Decidability of Bisimulation Equivalences for Parallel Timer Processes. In: Probst, D.K., von Bochmann, G. (eds.) *CAV 1992*. LNCS, vol. 663, pp. 302–315. Springer, Heidelberg (1993)
7. Corradini, F., Gorrieri, R., Roccati, M.: Performance Preorder: Ordering Processes with Respect to Speed. In: Hájek, P., Wiedermann, J. (eds.) *MFCS 1995*. LNCS, vol. 969, pp. 444–453. Springer, Heidelberg (1995)
8. 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)
9. 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)
10. 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)

11. Larsen, K.G., Behrmann, G., Bouyer, P., Pelanek, R.: Lower and upper bounds in zone-based abstractions of timed automata. *Int. J. Softw. Tools Technol. Transf.* 8, 204–215 (2006)
12. Geilen, M., Tripakis, S., Wiggers, M.: The earlier the better: a theory of timed actor interfaces. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, pp. 23–32 (2011)
13. Guha, S., Narayan, C., Arun-Kumar, S.: On decidability of prebisimulation for timed automata. Technical Report, Indian Institute of Technology Delhi, New Delhi, India (2012),
<http://www.cse.iitd.ernet.in/~shibashis/webpage/prebisim.pdf>
14. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. *Information and Computation* 111, 193–244 (1994)
15. Satoh, I.I., Tokoro, M.: A Formalism for Remotely Interacting Processes. In: Ito, T. (ed.) *TPPP 1994*. LNCS, vol. 907, pp. 216–228. Springer, Heidelberg (1995)
16. Larsen, K.G., Yi, W.: Time Abstracted Bisimulation: Implicit Specifications and Decidability. In: Main, M.G., Melton, A.C., Mislove, M.W., Schmidt, D., Brookes, S.D. (eds.) *MFPS 1993*. LNCS, vol. 802, pp. 160–176. Springer, Heidelberg (1994)
17. Milner, R.: *Communication and Concurrency*. Prentice Hall (1989)
18. Moller, F., Tofts, C.: Relating Processes with Respect to Speed. In: Groote, J.F., Baeten, J.C.M. (eds.) *CONCUR 1991*. LNCS, vol. 527, pp. 424–438. Springer, Heidelberg (1991)
19. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. *SIAM Journal on Computing* 16(6), 973–989 (1987)
20. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstraction bisimulations. *Formal Methods in System Design* 18, 25–68 (2001)
21. Weise, C., Lenzkes, D.: Efficient Scaling-Invariant Checking of Timed Bisimulation. In: Reischuk, R., Morvan, M. (eds.) *STACS 1997*. LNCS, vol. 1200, pp. 177–188. Springer, Heidelberg (1997)
22. Yannakakis, M., Lee, D.: An efficient algorithm for minimizing real-time transition systems. *Form. Methods Syst. Des.* 11, 113–136 (1997)

Exercises in Nonstandard Static Analysis of Hybrid Systems

Ichiro Hasuo¹ and Kohei Suenaga²

¹ University of Tokyo, Japan

² Kyoto University, Japan

Abstract. In formal verification of hybrid systems, a big challenge is to incorporate continuous *flow* dynamics in a discrete framework. Our previous work proposed to use *nonstandard analysis (NSA)* as a vehicle from discrete to hybrid; and to verify hybrid systems using a Hoare logic. In this paper we aim to exemplify the potential of our approach, through transferring *static analysis* techniques to hybrid applications. The transfer is routine via the *transfer principle* in NSA. The techniques are implemented in our prototype automatic precondition generator.

1 Introduction

Hybrid systems exhibit both discrete (digital) *jump* and continuous (physical) *flow* dynamics. They are of paramount importance now that so many physical systems—cars, airplanes, etc.—are controlled with computers. For an approach to hybrid systems from the formal verification community, the challenge is: 1) to incorporate flow-dynamics; and 2) to do so at the lowest possible cost, so that the discrete framework smoothly transfers to hybrid situations. A large body of existing work uses *differential equations* explicitly in the syntax; see the discussion of related work below.

In [22], instead, we proposed to introduce a constant dt for an *infinitesimal* (i.e. infinitely small) value, and *turn flow into jump*. With dt , the continuous operation of integration can be represented by a while-loop, to which existing discrete techniques such as Hoare-style program logics readily apply. For a rigorous mathematical development we employed *nonstandard analysis (NSA)* beautifully formalized by Robinson [16].

Concretely, in [22] we took the common triple of a WHILE-language, a first-order assertion language and a Hoare logic (e.g. in the textbook [23]); and added a constant dt to obtain a modeling and verification framework for hybrid systems. Its three components are called WHILE^{dt} , ASSN^{dt} and HOARE^{dt} . These are connected by denotational semantics defined in the language of NSA. We proved soundness and relative completeness of the logic HOARE^{dt} . Underlying the technical development is the idea of what we call *sectionwise execution*, illustrated by the following example.

Example 1.1 Let c_{elapse} be the program on the right. The value of dt is infinitesimal; therefore the while loop will not terminate within finitely many steps. Nevertheless it is somehow intuitive to expect that after an “execution” of this program, the value of t should be infinitesimally close to 1.

```
t := 0 ;
while t ≤ 1 do
    t := t + dt
```

Our idea is to think about *sectionwise execution*. For each natural number i we consider the i -th section of the program c_{elapse} , denoted by $c_{\text{elapse}}|_i$ and shown on the right. Concretely, $c_{\text{elapse}}|_i$ is obtained by replacing the infinitesimal dt in c_{elapse} with $\frac{1}{i+1}$. Informally $c_{\text{elapse}}|_i$ is the “ i -th approximation” of the original c_{elapse} .

A section $c_{\text{elapse}}|_i$ does terminate within finite steps and yields $1 + \frac{1}{i+1}$ as the value of t . Now we collect the outcomes of sectionwise executions and obtain a sequence

$$(1 + 1, 1 + \frac{1}{2}, 1 + \frac{1}{3}, \dots, 1 + \frac{1}{i}, \dots) \quad (1)$$

which is thought of as a progressive approximation of the actual outcome of the original program c_{elapse} . Indeed, in the language of NSA, the sequence (1) represents a *hyperreal number* r that is infinitesimally close to 1.

We note that a program in WHILE^{dt} is *not* executable in general: the program c_{elapse} executes infinitely many iterations. It is however a merit of *static* approaches to verification, that programs need not be executed to prove their correctness. Instead, well-defined mathematical semantics suffices and supports deductive reasoning. This is what we do, with the denotational semantics of WHILE^{dt} exemplified in Example (1).

This paper is a first step towards serious use of our framework of [22]. We employ various discrete strategies for reasoning about programs—in Hoare-style logics, the concern is mostly *invariant discovery*—and verify small, but nontrivial, examples. Such discrete techniques are actively pursued in program verification, or, in the *static analysis* community. This paper aims to exemplify our framework’s potential for transferring static analysis techniques to hybrid applications. In it we rely on nonstandard analysis; hence our venture is called *nonstandard static analysis*. We have implemented a prototype that generates a precondition, given a program and a postcondition.

In what follows we present the strategies for simplification and invariant/precondition discovery that we implemented, and prove their correctness. In fact we only present the strategies’ *standard* version—i.e. for WHILE and HOARE, as opposed to the *nonstandard* version that is for WHILE^{dt} and HOARE^{dt}. This is because the transfer from the former to the latter is routine work. Soundness of the nonstandard version follows from that of the standard one, almost trivially via the *sectionwise lemmas*. We will demonstrate this process using one of the strategies, in §4.2.

A related issue is the legitimacy of use of *Mathematica* for symbolic computation in our prototype. What we prove in *Mathematica* are formulas about real numbers, not hyperreals; and this needs justification. In §5.2 we address this issue and show that proofs in *Mathematica* indeed prove formulas about hyperreal numbers. The key is the *transfer principle*, a celebrated result in NSA (see §2 later).

We defer most of the proofs to the extended version (1).

Related Work. There have been extensive research efforts towards hybrid systems from the formal verification community. Unlike the current work where we turn flow into jump via dt, most of them feature acute distinction between flow- and jump-dynamics.

Model-checking approaches to hybrid systems have been studied for quite a while, with the successful formalism of *hybrid automaton* (1), on the one hand. On the other hand, deductive approaches have seen great advancement through a recent series of work by Platzer and his colleagues (including [13] [15]), resulting in the automated

prover KeYmaera. Our *nonstandard static analysis* approach currently falls much short of theirs in scalability and sophistication. However some of their techniques can be translated into the techniques in our framework; one example is the *differential invariant* strategy (§5.3). Interestingly in [15] it is argued that: being hybrid imposes no additional burden to verification in principle. This concurs with our claim.

Research in static analysis resulted in a huge number of verification techniques—[4, 7, 19] to name just a few. Some of them have been already used for hybrid applications (modeled with explicit differential equations) [17, 18, 20]. Our thesis is that these discrete techniques can be transferred to hybrid applications *as they are*, via NSA. In §4.1 we transfer the *phase split* technique in [21].

The use of NSA as a foundation of hybrid system modeling is not proposed for the first time; see e.g. [3, 5, 8]. Compared to this existing work, we claim our novelty is the use of NSA machinery (notably the transfer principle) in actual, automatic verification.

One recent research program (resulting e.g. in [6]) aims to employ *continuous* techniques—from the theory of dynamical systems—in purely discrete programs and applications. This is opposite to our approach (discrete techniques applied to continuous applications). We believe, however, that the two directions are not contending ones. It is not at all our intention here to champion the superiority of discrete techniques; our point instead is that the collection of available discrete techniques has much wider applicability. It is indeed our future work to *combine* discrete and continuous techniques.

2 Preliminaries

We summarize our previous work [22], focusing on providing intuitions. We denote the syntactic equality by \equiv .

Nonstandard Analysis. For detailed expositions see e.g. [9, 12].

We fix an *ultrafilter* $\mathcal{F} \subseteq \mathcal{P}(\mathbb{N})$ that extends the cofinite filter $\mathcal{F}_c := \{S \subseteq \mathbb{N} \mid \mathbb{N} \setminus S \text{ is finite}\}$. Its properties to be noted: 1) for any $S \subseteq \mathbb{N}$, exactly one of S and $\mathbb{N} \setminus S$ belongs to \mathcal{F} ; 2) if S is *cofinite* (i.e. $\mathbb{N} \setminus S$ is finite), then S belongs to \mathcal{F} .

Definition 2.1 (Hypernumber $d \in {}^*\mathbb{D}$) For a set \mathbb{D} (typically it is \mathbb{N} or \mathbb{R}), we define the set ${}^*\mathbb{D}$ by ${}^*\mathbb{D} := \mathbb{D}^\mathbb{N} / \sim_{\mathcal{F}}$. It is the set of infinite sequences on \mathbb{D} modulo the following equivalence $\sim_{\mathcal{F}}$: we define $(d_0, d_1, \dots) \sim_{\mathcal{F}} (d'_0, d'_1, \dots)$ by

$$\{i \in \mathbb{N} \mid d_i = d'_i\} \in \mathcal{F}, \quad \text{for which we say "}d_i = d'_i \text{ for almost every } i\text{".}$$

Therefore, given that two sequences $(d_i)_i$ and $(d'_i)_i$ coincide except for finitely many indices i , they represent the same hypernumber. Other predicates (such as $<$) are defined in the same way. For example a hyperreal $\omega^{-1} := [(1, \frac{1}{2}, \frac{1}{3}, \dots)]$ is positive ($0 < \omega^{-1}$) but is smaller than any (standard) positive real $r = [(r, r, \dots)]$.

The Framework of WHILE^{dt}, ASSN^{dt} and HOARE^{dt}. We take the standard combination (e.g. in [23]) of a while-style programming language WHILE, a first-order assertion language ASSN and a Hoare-style program logic HOARE (we equip them with constants for all real numbers). Then we augment the framework with a constant dt that denotes

a specific infinitesimal $\omega^{-1} = [(1, \frac{1}{2}, \frac{1}{3}, \dots)]$, and obtain the *nonstandard* framework consisting of WHILE^{dt}, ASSN^{dt} and HOARE^{dt}.

Definition 2.2 (WHILE^{dt}, WHILE, ASSN^{dt}, ASSN) The syntax of WHILE^{dt} is:

$$\begin{aligned}\mathbf{AExp} \ni a ::= & x \mid r \mid a_1 \text{aop} a_2 \mid \text{dt} \\ & \text{where } x \in \mathbf{Var}, r \text{ is a constant for } r \in \mathbb{R}, \text{ and aop} \in \{+, -, \cdot, /, \lceil _ \rceil\} \\ \mathbf{BExp} \ni b ::= & \text{true} \mid \text{false} \mid b_1 \wedge b_2 \mid \neg b \mid a_1 < a_2 \\ \mathbf{Cmd} \ni c ::= & \text{skip} \mid x := a \mid c_1; c_2 \mid \text{assert } b \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c\end{aligned}$$

Here **AExp** is the set of *arithmetic expressions*; **BExp** and **Cmd** are those of *Boolean* and *command* expressions. The operator $\lceil r \rceil$ denotes the smallest natural number which is larger than or equal to the real number r . We will use this operator in §5.4. The command assert b amounts to skip if b is true; an infinite loop (divergence) otherwise.

Our first-order assertion language¹ ASSN^{dt} consists of the following *formulas*.

$$\mathbf{Fml} \ni A ::= \text{true} \mid \text{false} \mid A_1 \wedge A_2 \mid \neg A \mid a_1 < a_2 \mid \forall x \in {}^*\mathbb{N}. A \mid \forall x \in {}^*\mathbb{R}. A \quad \text{where } x \in \mathbf{Var} \text{ and } a_i \in \mathbf{AExp}$$

By WHILE, we denote the fragment of WHILE^{dt} without the constant dt.

By ASSN we designate the language obtained from ASSN^{dt} by: 1) dropping the constant dt; and 2) replacing the quantifiers $\forall x \in {}^*\mathbb{N}$ and $\forall x \in {}^*\mathbb{R}$ with $\forall x \in \mathbb{N}$ and $\forall x \in \mathbb{R}$, respectively, i.e. with those which range over standard numbers.

It is essential that in ASSN^{dt} we allow only *hyperquantifiers* $\forall x \in {}^*\mathbb{R}$ and not *standard* ones $\forall x \in \mathbb{R}$. This is much like with the *transfer principle* in NSA [12 Thm. II.4.5].

Definition 2.3 (Section $e|_i$) Let e be an expression of WHILE^{dt} or ASSN^{dt}, and $i \in \mathbb{N}$. The i -th section of e , denoted by $e|_i$, is obtained by: 1) replacing every occurrence of dt with the constant $1/(i+1)$; and 2) replacing every hyperquantifier $\forall x \in {}^*\mathbb{D}$ with $\forall x \in \mathbb{D}$. Here $\mathbb{D} \in \{\mathbb{N}, \mathbb{R}\}$. Obviously a section $e|_i$ is an expression of WHILE or ASSN.

Definition 2.4 (HOARE^{dt}, HOARE) HOARE^{dt} is a system that derives *Hoare triples* $\{A\}c\{B\}$ (a triple of ASSN^{dt} formulas A, B and a WHILE^{dt} command c) using the following rules. The rules are the same for HOARE.

$$\begin{array}{ccl} \overline{\{A\} \text{skip } \{A\}} & (\text{SKIP}) & \overline{\{A[a/x]\} x := a \{A\}} & (\text{ASSIGN}) \\ \overline{\{A\} c_1 \{C\} \quad \{C\} c_2 \{B\}} & (\text{SEQ}) & \overline{\{A \wedge b\} c_1 \{B\} \quad \{A \wedge \neg b\} c_2 \{B\}} & (\text{IF}) \\ \overline{\begin{array}{c} \{A\} c_1; c_2 \{B\} \\ \{A \wedge b\} c \{A\} \end{array}} & & \overline{\begin{array}{c} \{A\} \text{if } b \text{ then } c_1 \text{ else } c_2 \{B\} \\ \models A \Rightarrow A' \quad \{A'\} c \{B'\} \quad \models B' \Rightarrow B \end{array}} & (\text{CONSEQ}) \\ \overline{\{A\} \text{while } b \text{ do } c \{A \wedge \neg b\}} & (\text{WHILE}) & & \overline{\{A\} c \{B\}} \\ \hline \overline{\{b \Rightarrow A\} \text{assert } b \{A\}} & (\text{ASSERT}) & & \end{array}$$

We write $\vdash \{A\}c\{B\}$ if the triple $\{A\}c\{B\}$ can be derived using the above rules.

¹ The term *assertion language* has little to do with the command assert b in WHILE^{dt}.

We turn to semantics. Denotational semantics of WHILE^{dt} is defined following the intuition in Example 1.1. There the semantics $\llbracket c \rrbracket$ of a command c is a *hyperstate transformer* that maps a hyperstate σ (a state that stores hypernumbers) to a hyperstate $\llbracket c \rrbracket \sigma$.

Semantics of an assertion $A \in \mathbf{Fml}$ is defined in the usual way; we write $\models A$ and say A is *valid* if $\sigma \models A$ for each hyperstate σ . A Hoare triple is *valid* ($\models \{A\}c\{B\}$) if, for any hyperstate σ such that $\sigma \models A$, we have $\llbracket c \rrbracket \sigma \models B$.

Three *sectionwise lemmas* play central roles in our framework. They correspond to *Łoś's theorem* in NSA. We only present the following one, that is the most relevant to the current paper. Recall the meaning of “for almost every i ” via an ultrafilter (§2).

Lemma 2.5 (Sectionwise validity of Hoare triples) *Let A, B be ASSN^{dt} formulas, and $c \in \mathbf{Cmd}$ be a WHILE^{dt} command. We have*

$$\models \{A\}c\{B\} \iff \models \{A|_i\} c|_i \{B|_i\} \text{ for almost every } i. \quad \square$$

Definition 2.6 (*-transform) Let A be an ASSN formula. We define its **-transform*, denoted by *A , to be the ASSN^{dt} formula obtained from A by replacing every occurrence of a standard quantifier $\forall x \in \mathbb{D}$ with the corresponding hyperquantifier $\forall x \in {}^*\mathbb{D}$.

Proposition 2.7 (Transfer principle) 1. For each ASSN formula A , $\models A$ iff $\models {}^*A$.
 2. For any dt-free ASSN^{dt} formula A , the following are equivalent: 1) $\models A|_i$ for each $i \in \mathbb{N}$; 2) $\models A|_i$ for some $i \in \mathbb{N}$; 3) $\models A$. \square

3 A Leading Example: ETCS

We use verification of (a simplified version of) the *European Train Control System (ETCS)* as a leading example—this is done also in many chapters of [14]. We go step by step, introducing the strategies as the need arises.

Our target program ETCS₀ is shown on the right. It contains small fragments of the European Train Control System (ETCS). Here a , t , v , and z are variables that represent acceleration rate, time, velocity, and position, respectively. The symbols m , s , b , a_0 , eps are all constants, all of which are assumed to be (strictly) positive. Here, m is the position beyond which the train must not run (“a wall”); s is the *safety distance* such that, once the train’s distance from the wall is less than s , it starts braking; b is the rate used in braking; a_0 is the (positive) acceleration rate used unless the train is braking. The check if $m - z < s$ occurs once every eps seconds.

The (post)condition to be guaranteed after the execution of ETCS₀ is the safety property $z < m$; it is inserted in Fig. II as an annotation, between (: and :). We set out to calculate a precondition—a relationship among constants as well as (the initial values of) the variables—that ensures this postcondition.

The program ETCS₀ is itself a while-loop; thus to calculate a precondition we need to discover a (loop) invariant. However the loop is a complicated one with an if branch and another while-loop inside. We first preprocess ETCS₀ and simplify it, employing some program transformation strategies studied in the field of static analysis.

```

while (v > 0) {
    if m - z < s
        then a := -b
        else a := a0;
    t := 0;
    while (t < eps && v > 0) {
        z := z + v * dt;
        v := v + a * dt;
        t := t + dt
    }
    (: z < m :)
  
```

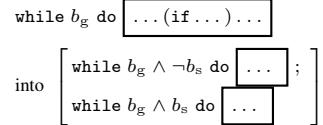
Fig. 1. The original program ETCS₀

4 Program Simplification Strategies

4.1 Phase Split

Looking at ETCS₀ (Fig. 1), we can imagine that once the train starts braking, it never starts to accelerate. That is, we could transform the outer loop in ETCS₀ into two successive loops, with the former loop solely for accelerating and the latter loop for braking.

It is the program transformation technique proposed in [21] (closely related to those in [2, 10]) that makes the above intuition rigorous. It eliminates an if-branch inside a while loop using a *phase splitter predicate* b_s , as is shown on the right. This simplification strategy shall be referred to as *phase split*.



First we follow [21] and describe the strategy formalized in WHILE and HOARE. In §4.2 we use the sectionwise lemmas and show that the nonstandard framework (WHILE^{dt} and HOARE^{dt}), too, admits the same strategy.

Definition 4.1 (Holed command, pre-hole fragment) The set $\mathbf{Cmd}_{[_]}$ of *holed commands* of WHILE—those which contain exactly one hole $[_]$ for a guard of if—is defined by the BNF expression below. Here $c, c_1, c_2 \in \mathbf{Cmd}$ are commands (without holes). The result of replacing $[_]$ with $b \in \mathbf{BExp}$ in $h \in \mathbf{Cmd}_{[_]}$ is denoted by $h[b]$.

$$\mathbf{Cmd}_{[_]} \ni h ::= \begin{array}{l} \text{if } [_] \text{ then } c_1 \text{ else } c_2 \mid h; c \mid c; h \\ \quad \text{if } b \text{ then } h \text{ else } c \mid \text{if } b \text{ then } c \text{ else } h \end{array}$$

$$\frac{\text{if } [_] \text{ then } c_1 \text{ else } c_2 \ : \equiv \ \text{skip}}{\text{if } b \text{ then } h \text{ else } c \ : \equiv \ \text{assert } b ; \overline{h}} \quad \frac{\overline{h}; c \ : \equiv \ \overline{h} \quad \overline{c; h} \ : \equiv \ c; \overline{h}}{\text{if } b \text{ then } c \text{ else } \overline{h} \ : \equiv \ \text{assert } \neg b ; \overline{h}}$$

For each holed command h , its *pre-hole fragment* \overline{h} is defined inductively as above. Intuitively, it is the fragment of h that is executed before hitting the hole $[_]$.

Lemma 4.2 (Phase split [21]) If a Boolean expression $b_s \in \mathbf{BExp}$ satisfies

$$\models \{b_s\} \overline{h} \{b_c\}, \quad \models \{\neg b_s\} \overline{h} \{\neg b_c\}, \quad \text{and} \quad \models \{b_g \wedge b_s\} h[b_c] \{\neg b_g \vee b_s\}, \quad (2)$$

then we have $\llbracket c_0 \rrbracket = \llbracket c_1 \rrbracket$ between the commands

$$\begin{array}{l} c_0 \ : \equiv \ \text{while } b_g \text{ do } h[b_c], \quad \text{and} \\ c_1 \ : \equiv \ [\text{while } (b_g \wedge \neg b_s) \text{ do } h[\text{false}] ; \text{while } (b_g \wedge b_s) \text{ do } h[\text{true}]] \ . \end{array} \quad (3) \quad \square$$

Such b_s is called a *phase splitter predicate*. In applying this lemma to ETCS₀, an obvious candidate for a phase splitter predicate is $m - z < s$, the guard of the if-branch itself. (Its negation $m - z \geq s$ is a candidate too; but the side conditions (2) cannot be discharged.)

To discharge the side conditions (2) is not hard: in its course we use the *differential invariant* strategy described later in §5.3. We further apply obvious simplifications, such as $\llbracket \text{if true then } c_1 \text{ else } c_2 \rrbracket = \llbracket c_1 \rrbracket$ and constant propagation; consequently we are led to the program ETCS₁ (on the right) that is equivalent to ETCS₀.

```

while (v > 0 && m - z >= s) {
  a := a0; t := 0;
  while (t < eps && v > 0) {
    z := z + v * dt;
    v := v + a0 * dt;
    t := t + dt );
  }
  while (v > 0 && m - z < s) {
    a := -b; t := 0;
    while (t < eps && v > 0) {
      z := z + v * dt;
      v := v - b * dt;
      t := t + dt );
    }
    (: z < m :)
  }
}

```

Fig. 2. The program ETCS₁

4.2 From Standard to Nonstandard I: Modular Transfer

As stated in the introduction, the current paper presents strategies (for simplification and invariant/precondition discovery) only for the standard framework (WHILE and HOARE), leaving out the corresponding nonstandard strategies (for WHILE^{dt} and HOARE^{dt}). This is because the transfer from the former to the latter is straightforward. Here we describe the process of such transfer, with the phase split strategy (§4.1) as an example. Here the *sectionwise lemmas* (in particular Lem. 2.5) play the key role.

The syntactic notions of holed command and pre-hole fragment are defined in WHILE^{dt}, in the exactly the same way as in WHILE (Def. 4.1).

Lemma 4.3 (Nonstandard phase split) *Assume $b_s \in \mathbf{BExp}$ satisfies, in HOARE^{dt},*

$$\models \{b_s\} \overline{h} \{b_c\}, \quad \models \{\neg b_s\} \overline{h} \{\neg b_c\}, \quad \text{and} \quad \models \{b_g \wedge b_s\} h[b_c] \{\neg b_g \vee b_s\}.$$

Let c_0 and c_1 be as in (3), now in WHILE^{dt}. We have $\llbracket c_0 \rrbracket = \llbracket c_1 \rrbracket$. \square

Proof. Let $\sigma \in \mathbf{HSt}$ be a hyperstate; and $(\sigma|_i)_{i \in \mathbb{N}}$ be its arbitrary sequence representation. By the definition of hypernumbers, it suffices to show that

$$\llbracket c_0|_i \rrbracket(\sigma|_i) = \llbracket c_1|_i \rrbracket(\sigma|_i) \quad \text{for almost every } i. \quad (4)$$

By the assumptions and Lem. 2.5, each one of the following holds for almost every i .

$$\models \{b_s|_i\} \overline{h}|_i \{b_c|_i\} \models \{(\neg b_s)|_i\} \overline{h}|_i \{(\neg b_c)|_i\} \models \{(b_g \wedge b_s)|_i\} h[b_c]|_i \{(\neg b_g \vee b_s)|_i\}.$$

Therefore, by the definition of ultrafilter, all three of the above hold simultaneously, for almost every i . That is, for almost every i , we have all of the following three hold:

$$\models \{b_s|_i\} \overline{h}|_i \{b_c|_i\} \models \{\neg(b_s|_i)\} \overline{h}|_i \{\neg(b_c|_i)\} \models \{b_g|_i \wedge b_s|_i\} h|_i[b_c|_i] \{\neg(b_g|_i) \vee b_s|_i\}$$

where we also used an obvious fact $\overline{h}|_i \equiv (\overline{h|_i})$. To each such i we apply the standard version of the strategy (Lem. 4.2) and obtain, for almost every i ,

$$\begin{aligned} & \llbracket \mathbf{while } b_g|_i \mathbf{do } h|_i[b_c|_i] \rrbracket(\sigma|_i) \\ &= \llbracket \mathbf{while } (b_g|_i \wedge \neg b_s|_i) \mathbf{do } h|_i[\mathbf{false}] ; \mathbf{while } (b_g|_i \wedge b_s|_i) \mathbf{do } h|_i[\mathbf{true}] \rrbracket(\sigma|_i) \end{aligned}$$

Using another obvious fact that $h[b_s]|_i \equiv (h|_i)[b_s|_i]$, this is equivalent to (4). \square

Note the *modularity* in the proof: the standard version's proof (Lem. 4.2) is completely hidden. For any other (standard) strategy presented in this paper, its nonstandard version follows in the same way via sectionwise arguments.

4.3 Superfluous Guard Elimination

In ETCS₁ each while-loop has its guard consisting of two atomic inequalities. This causes problems with our toolkit for invariant/precondition discovery, especially with the *counting iterations* strategy (§5.4). In fact, some of the conditions are seemingly superfluous: for example, in the second (internal) while-loop, $m - z < s$ is an invariant.

Lemma 4.4 (Superfluous guard elimination) *For commands*

$$\begin{aligned} c_0 &:= \text{while } (b \wedge b_{sf}) \text{ do } c, \text{ and} \\ c_1 &:= \text{if } b_{sf} \text{ then } (\text{while } b \text{ do } c) \text{ else skip ,} \end{aligned}$$

if we have $\models \{b \wedge b_{sf}\} c \{b_{sf}\}$, then $\llbracket c_0 \rrbracket = \llbracket c_1 \rrbracket$. \square

This strategy, together with some straightforward simplifications, transforms ETCS₁ into ETCS₂ shown below. For each of the three instances of the strategy, the side condition is easily discharged using the *differential invariant* strategy (§5.3).

The program

ETCS₂:

```
if (v > 0)
  then
    while (m - z >= s) {
      a := a0; t := 0;
      while (t < eps) {
        z := z + v * dt;
        v := v + a0 * dt;
        t := t + dt }
      else skip;
```

```
while (v > 0) {
  a := -b; t := 0;
  while (t < eps && v > 0) {
    z := z + v * dt;
    v := v - b * dt;
    t := t + dt }
  (: z < m :)
```

4.4 Time Elapse

The challenge in verifying WHILE programs is in while-loops. The second half of ETCS₂ is a loop with another loop inside; we wish to simplify this.

A close look reveals that the inside loop is vacuous: since the inside guard and the outside guard share the same condition $v > 0$, the condition $t < \text{eps}$ has in fact no effect.² We often encounter this situation in the verification of WHILE^{dt} programs—other simplification strategies easily lead to such vacuous nested loops.

Lemma 4.5 (Time elapse) *Let t be a variable that does not occur in $b \in \mathbf{BExp}$ or $c \in \mathbf{Cmd}$; ε be a positive constant; and*

$$\begin{aligned} c_0 &:= \text{while } b \text{ do } (t := 0; \text{while } t < \varepsilon \text{ do } (t := t + dt; c)) , \\ c_1 &:= \text{while } b \text{ do } c . \end{aligned}$$

Then the denotations $\llbracket c_0 \rrbracket$ and $\llbracket c_1 \rrbracket$ coincide on all variables but t . \square

Lem. 4.5 allows several straightforward generalizations. Currently we do not need them.

Since t does not occur in the postcondition $z < m$, Lem. 4.5 ensures that *time elapse* is a sound transformation. It simplifies the second half of ETCS₂ into ETCS₃ below.

The program

ETCS₃:

```
if (v > 0)
  then
    while (m - z >= s) {
      a := a0; t := 0;
      while (t < eps) {
        z := z + v * dt;
        v := v + a0 * dt;
        t := t + dt }
      else skip;
```

```
while (v > 0) {
  a := -b;
  z := z + v * dt;
  v := v - b * dt ;
  (: z < m :)
```

² This is not the case with the first half of ETCS₂: it might be $m - z < s$ but still the train can accelerate for eps seconds. Therefore the nested loops cannot be suppressed so easily.

5 Precondition/Invariant Discovery Strategies

We now describe three strategies aimed at the discovery of suitable preconditions/invariants for while-loops. In fact they return directly a precondition (when they succeed).

We desire their output to be quantifier-free, since quantifiers often incur prohibiting performance penalties. This is what we do in our prototype: quantifiers appear only in the *Mathematica* backend. In what follows several quantifiers do occur; but they are in the correctness proofs of the strategies, i.e. on the meta level.

5.1 Invariant via Quantifier Elimination

We now compute a precondition *precond1* that, after the execution of the last while-loop in *ETCS₃* (which we denote by *c_{brake}*), ensures the postcondition $z < m$ hold.

Recall our thesis: via nonstandard analysis, we can represent continuous flow by discrete jumps, so that (discrete) verification techniques readily apply. However, once we cast a *continuous* look at *c_{brake}*, it obviously represents a simple flow dynamics governed by the differential equations $\dot{v}(t) = \ddot{z}(t) = -b$. In this “continuous” perspective, one would solve the current question in the following way.

- First the differential equations are solved analytically, obtaining $v(t) = v_0 - bt$ and $z(t) = z_0 + v_0 t - \frac{1}{2}bt^2$. Here v_0 and z_0 are initial values.
- Using these analytic solutions, the desired precondition is nothing other than

$$\forall t \in \mathbb{R}. \left(v_0 - bt > 0 \wedge t \geq 0 \implies z_0 + v_0 t - \frac{1}{2}bt^2 < m \right). \quad (5)$$

One would then apply *quantifier elimination* and obtain a quantifier-free formula that is equivalent to (5). Some algorithms are known including *cylindrical algebraic decomposition (CAD)*. In *Mathematica* *Resolve* offers this functionality; it returns

$$v_0 \leq 0 \vee (z_0 < m \wedge 2bm - v_0^2 - 2bz_0 \geq 0).$$

This procedure of “first solve differential equations, then eliminate quantifiers” is common in the deductive verification of hybrid systems, such as in Platzer’s [14].

The next strategy (Lem. 5.2) is a discrete variation of this procedure, where differential equations are replaced with *difference equations* (also called *recurrence relations*).

We need preparation. In Lem. 5.2 we need to have a formula $A \overbrace{[a/x][a/x] \cdots [a/x]}^{y \text{ times}}$ as a formula with y as a free variable. This is not automatic—note that substitution $[a/x]$ and the number y live on the meta level. However, let us say that A is a formula $A \equiv (x > 0)$ and $a \equiv x + 1$. Then we have, for any $y \in \mathbb{N}$,

$$\models (x > 0) [x + 1/x] [x + 1/x] \cdots [x + 1/x] \Leftrightarrow x + y > 0,$$

where substitution is repeated y times. Thus the formula $x + y > 0$ effectively represents the formula $(x > 0) [x + 1/x] [x + 1/x] \cdots [x + 1/x]$. In other words, $x + y > 0$ is a *homogeneous representation* of substitutions $x > 0, x + 1 > 0, (x + 1) + 1 > 0, \dots$.

The general definition is as follows. There A_{rs} corresponds to $x + y > 0$ in the above example; it is a formula that must be discovered somehow. In our prototype it is done by solving a recurrence relation.

Definition 5.1 (Repeated substitution) Let x be a variable, A be a ASSN formula, a be an arithmetic expression, and y be a variable not occurring in A , a or x . A formula A_{rs} is said to be a *homogeneous representation* of A and $[a/x]$ if the following holds.

$$\models A_{\text{rs}}[0/y] \Leftrightarrow A \quad \wedge \quad \forall u \in \mathbb{N}. (A_{\text{rs}}[u+1/y] \Leftrightarrow (A_{\text{rs}}[u/y])[a/x]) \quad (6)$$

We abuse notations by denoting such A_{rs} by $A[a/x]^y$.

Lemma 5.2 (QE invariant)

$$\models \left\{ \begin{array}{l} (\neg b \Rightarrow A) \wedge \\ \forall y \in \mathbb{N}. ((b[a/x]^y \wedge \neg b[a/x]^{y+1}) \Rightarrow A[a/x]^{y+1}) \end{array} \right\} \text{while } b \text{ do } x := a \quad \{A\} .$$

The lemma is presented in the simplest form: it is straightforward to allow a sequence $x_1 := a_1; \dots; x_n := a_n$ of assignments inside the loop.

Proof. Let P be the precondition $(\neg b \Rightarrow A) \wedge \forall y \in \mathbb{N}. ((b[a/x]^y \wedge \neg b[a/x]^{y+1}) \Rightarrow A[a/x]^{y+1})$. By the (WHILE) rule of HOARE, it suffices to show that: 1) P is a loop invariant, that is, $\models \{P \wedge b\}x := a\{P\}$; and 2) P is strong enough, i.e. $\models P \wedge \neg b \Rightarrow A$.

To prove 1), assume $\sigma \models P \wedge b$. The goal is $\llbracket x := a \rrbracket \sigma \models P$, which is equivalent to $\sigma \models P[a/x]$, since $P[a/x]$ is the weakest precondition for $x := a$ and the postcondition P . This is further reduced to: $\sigma \models b[a/x]^{n+1} \wedge \neg b[a/x]^{n+2} \Rightarrow A[a/x]^{n+2}$ for any $n \in \mathbb{N}$ (which follows from $\sigma \models P$); and $\sigma \models \neg b[a/x] \Rightarrow A[a/x]$ (which follows from $\sigma \models b$ and $\sigma \models (b \wedge \neg b[a/x]) \Rightarrow A[a/x]$).

The condition 2) is obvious since $\models P \Rightarrow (\neg b \Rightarrow A)$. This concludes the proof. \square

In our prototype, we first compute homogeneous representations $b[a/x]^y$ and $A[a/x]^y$ by solving recurrence relations in y such as $A[a/x]^{y+1} = (A[a/x]^y)[a/x]$. Using this we form the precondition in Lem. 5.2. Then we further eliminate the quantifier $\forall y \in \mathbb{N}$ and obtain a quantifier-free precondition.

For example, in the case of c_{brake} in ETCS₃, we discover a homogeneous representation $A[\vec{d}/\vec{x}]^y \equiv \frac{1}{2}(bdt^2y^2 + 2dtvy - bdt^2y^2 + 2z) < m$. As a quantifier-free precondition we obtain the following, which is henceforth denoted by P_1 .

$$P_1 := (v > 0 \vee m > z) \wedge (b^2 dt^2 + 4bdtv + 8bz + 4v^2 < 8bm \vee bdtv + 2bz + v^2 \leq 2bm) \quad (7)$$

This strategy (Lem. 5.2) has the merit of having no side conditions. But in our experience it is computationally expensive: a slightly complicated postcondition (say, a combination of a few inequalities) makes the quantifier elimination step infeasible. In fact it does not currently work for the loops in ETCS₃ other than the last one c_{brake} .

5.2 From Standard to Nonstandard II: The Transfer Principle and Symbolic Computation in *Mathematica*

Here again we demonstrate how a standard strategy can be transferred to a nonstandard one. The strategy *QE invariant* (Lem. 5.2) employs *Mathematica* in two stages:

solving recurrence relations and eliminating quantifiers, both done over standard numbers (instead of hyperreals). Our focus is therefore on how this can be justified.

Definition 5.3 (Repeated substitution in ASSN^{dt}) Let x, A, a, y be as in Def. 5.1 but now in ASSN^{dt} . A formula A'_{rs} is said to be a *homogeneous representation* of A and $[a/x]$, and is denoted by $A[a/x]^y$, if the following holds.

$$\models A'_{\text{rs}}[0/y] \Leftrightarrow A \quad \wedge \quad \forall u \in {}^*\mathbb{N}. (A'_{\text{rs}}[u+1/y] \Leftrightarrow (A'_{\text{rs}}[u/y])[a/x]) \quad (8)$$

The sole difference from Def. 5.1 is the quantifier (ranging over hypernatural numbers).

How do we find A'_{rs} that satisfies (8)? We rely on the following lemma.

Lemma 5.4 *In the setting of Def. 5.3 let A' and a' be the expressions obtained from A and a by replacing the constant dt by a fresh variable d (and also by making hyperquantifiers standard). If A_{rs} is a homogeneous representation $A'[a'/x]^y$ in ASSN (Def. 5.1), then ${}^*A_{\text{rs}}[dt/d]$ is a homogeneous representation $A_{\text{rs}}[a/x]^y$ in ASSN^{dt} . \square*

Therefore it essentially suffices to prove (6). This is in ASSN hence *Mathematica* is capable of proving it.

Lemma 5.5 (Nonstandard QE invariant) *We have, in HOARE^{dt} ,*

$$\models \left\{ \begin{array}{l} (\neg b \Rightarrow A) \wedge \\ \forall y \in {}^*\mathbb{N}. ((b[a/x]^y \wedge \neg b[a/x]^{y+1}) \Rightarrow A[a/x]^{y+1}) \end{array} \right\} \text{while } b \text{ do } x := a \quad \{A\} .$$

Proof. By Lem. 2.5, the following suffices: in HOARE ,

$$\models \left\{ \left(\begin{array}{l} (\neg b \Rightarrow A) \wedge \\ \forall y \in {}^*\mathbb{N}. ((b[a/x]^y \wedge \neg b[a/x]^{y+1}) \Rightarrow A[a/x]^{y+1}) \end{array} \right) \middle|_i \right\} x := a|_i \quad \{A|_i\}$$

holds for almost every $i \in \mathbb{N}$. By Def. 5.3, Def. 5.1 and [22, Lem. 4.5], it is easy that $\models (A[a/x]^y)|_i \Leftrightarrow A|_i[a|_i/x]^y$ for almost every $i \in \mathbb{N}$. Therefore it suffices to show

$$\models \left\{ \begin{array}{l} (\neg b|_i \Rightarrow A|_i) \wedge \\ \forall y \in \mathbb{N}. ((b|_i[a|_i/x]^y \wedge \neg b|_i[a|_i/x]^{y+1}) \Rightarrow A|_i[a|_i/x]^{y+1}) \end{array} \right\} x := a|_i \quad \{A|_i\}$$

for almost every i . The last statement (in HOARE) is Lem. 5.2 itself. \square

The second use of *Mathematica*—in quantifier elimination—is justified much like in Lem. 5.4, via the transfer principle.

5.3 Differential Invariant

Most of the simplification strategies in §4 come with side conditions to be discharged. In many such cases, the postcondition to be established is itself an invariant. An example is the condition $v > 0$ in the first half of ETCS₁.

The following lemma provides an efficient method for proving such Hoare triples.³

³ The name is taken from Platzer's extensive treatment of a similar notion [14]. The correspondence is as follows: when the while-loop below represents continuous dynamics, the condition $a_c[a/x] < a_c$ means that the first derivative of a_c is negative.

Lemma 5.6 (Differential Invariant) Assume that an arithmetic expression a_c satisfies $\models b \Rightarrow a_c[a/x] < a_c$. Then $\models \{a_c < 0\} \text{ while } b \text{ do } x := a \{a_c < 0\}$. \square

In our current prototype a candidate for the invariant $a_c < 0$ is chosen simply out of the disjuncts of the postcondition. More sophisticated candidate generation methods—such as in [14]—are left as future work.

5.4 Iteration Count

Verification of the leading example ETCS (§3) has been reduced, in §5.1, to the situation below on the left. Here postcondition P_1 is as in (7). Our prototype further simplifies the situation into the one on the right.

<pre> if (v > 0) then while (m - z >= s) { a := a0; t := 0; while (t < eps) { z := z + v * dt; v := v + a0 * dt; t := t + dt } else skip (: P_1 :)</pre>	<pre> while (m - z >= s) { a := a0; t := 0; while (t < eps) { z := z + v * dt; v := v + a0 * dt; t := t + dt } (: P_2 :) where P_2 := b^2 dt^2 + 4bdv + 8bz + 4v^2 < 8bm ∨ bdv + 2bz + v^2 ≤ 2bm (9)</pre>
---	--

We now face a nested loop to which none of the previous strategies applies. Our last strategy estimates the number of iterations in each loop—such as eps/dt for the inner loop⁴—and approximates commands by repeated applications of assignments. The strategy involves nontrivial side conditions but we experienced its wide applicability.

The formalization of the full *iteration count* strategy is cumbersome and buries its idea in overwhelming details. Here we only present the following restricted version that deals with a single loop. The full version that applies to nested loops is deferred to [11].

Lemma 5.7 Let a_i be an arithmetic expression. Assume the following conditions.

1. $\models \neg b[a/x]^{a_i}$
2. For some fresh variable y , $\models \{\neg b[a/x]^{y+1}\} c \{\neg b[a/x]^y\}$
3. $\models \forall u, v \in \mathbb{R}. ((a_i \leq u < v < a_i + 1 \wedge \neg b[a/x]^u) \Rightarrow \neg b[a/x]^v)$
4. For some fresh variable y , $\models \{A[a/x]^{y+1}\} c \{A[a/x]^y\}$
5. $\models \forall u, v \in \mathbb{R}. ((u < v \wedge A[a/x]^v) \Rightarrow A[a/x]^u)$

Then we have

$$\models \{A[a/x]^{a_i+1} \wedge a_i \geq 0\} \text{ while } b \text{ do } c \{A\} . \quad (10)$$

\square

Here are some intuitions. The expression a_i denotes an estimated number of iterations: for example, $a_i \equiv \text{eps}/dt$ for the inner loop on the right in (9). The estimation is done by solving the equation $a_i \cdot dt = \text{eps}$. We are also *simulating* execution of c by a

⁴ Note that the number eps/dt makes sense in NSA: it means $(i + 1) \cdot \text{eps}$ in the i -th section.

substitution $[a/x]$; the conditions ② and ④ assert that this simulation is sound. Note that these conditions are trivially satisfied if c is an assignment command $x := a$.

The conditions ③ and ⑤ impose suitable *monotonicity* requirements on the guard b and the postcondition A . Such monotonicity is needed for the following reason. The estimated number a_i of iterations need not be exactly some (hyper)natural number—it can lie between two (hyper)natural numbers, i.e. $\lceil a_i \rceil - 1 < a_i < \lceil a_i \rceil$. This is why we are *conservative* in ⑩; we use $a_i + 1$, instead of a_i , as a number of iterations. The monotonicity conditions ③ & ⑤ ensure that this conservative approximation is sound.

The full version of this *iteration count* strategy (Lem. 5.7) succeeds in the remaining bit of the leading example (on the right in ⑨). Its output is a Boolean combination of inequalities; we denote this formula by P_3 . It is complicated—not least due to occurrences of dt —and we would rather not write it down explicitly.

5.5 Cast to Shadow

We have established that $\models \{P_3\} \text{ETCS}_0 \{z < m\}$. As the last step, we wish to eliminate the occurrences of dt in the precondition P_3 . This results in a much simpler precondition. The intuition is: the condition $\varepsilon + dt < 0$ is implied by $\varepsilon < 0$ if ε is a constant denoting a standard real number. To put it in rigorous terms:

Lemma 5.8 (Cast to Shadow) *Let a be an arithmetic expression in WHILE^{dt} , d be a fresh variable, and $a[d/dt]$ be the expression in WHILE obtained by substituting d for dt in a . Assume the following.*

1. *The expression a is closed, that is, it has no occurrences of variables.*
2. *Let σ be a (standard) state that is not \perp ; it determines a function⁵ $f_\sigma : \mathbb{R} \rightarrow \mathbb{R}$ by $r \mapsto \llbracket a[d/dt] \rrbracket(\sigma[d \mapsto r])$. We assume that f_σ is continuous at $r = 0$ for any given $\sigma \neq \perp$. Here $\llbracket a[d/dt] \rrbracket$ is defined by denotational semantics [22 §3.2]; and $\sigma[d \mapsto r]$ is an updated state.*

Then $\models (a[d/dt])[0/d] < 0 \implies a < 0$. □

Note that, by the definition of $a[d/dt]$, the expression $(a[d/dt])[0/d]$ is the result of replacing dt in a with 0—i.e. of “eliminating dt .” In using the lemma, the closedness assumption (Cond. ①) can be enforced by setting the initial values of variables as constants (that necessarily represent standard reals by the definition of WHILE^{dt}). The continuity assumption (Cond. ②) in the lemma is most of the time satisfied since the arithmetic operations of WHILE are all continuous except for zero-division.

In nonstandard analysis, a hyperreal number r' that is not infinite has a unique (standard) real number r that is infinitesimally close to r' (we write $r \simeq r'$). Such r is called the *shadow* of r' ; hence the name of the strategy.

This strategy can be used for simplifying the precondition P_3 (or, more generally, a Boolean combination of inequalities) in the following way.

- First, P_3 is converted to CNF (or DNF; it does not matter): $\models P_3 \Leftrightarrow \bigwedge_i \bigvee_j L_{i,j}$. Note that every literal $L_{i,j}$ occurs positively.

⁵ In general a partial function since zero-division might occur.

- Among all the literals $L_{i,j}$, the negative ones (like $L_{i,j} \equiv \neg(a_{i,j} < b_{i,j})$) is turned into a positive one (like $a_{i,j} \geq b_{i,j}$), by reversing inequalities.
- Each inequality is turned into the form of either $c_{i,j} < 0$ or $c_{i,j} \leq 0$. For example: we turn $a_{i,j} \geq b_{i,j}$ into $b_{i,j} - a_{i,j} \leq 0$. Thus we have obtained $\models P_3 \Leftrightarrow \bigwedge_i \bigvee_j c_{i,j} \triangleleft_{i,j} 0$, where each $\triangleleft_{i,j}$ is either $<$ or \leq .
- Obviously $\models c_{i,j} < 0 \Rightarrow c_{i,j} \triangleleft_{i,j} 0$; thus $\models \bigwedge_i \bigvee_j c_{i,j} < 0 \Rightarrow P_3$.
- Finally, to each inequality $c_{i,j} < 0$ we apply Lem. 5.8 (given that the side conditions are discharged). This establishes $\models \bigwedge_i \bigvee_j (c_{i,j}[d/dt])[0/d] < 0 \Rightarrow P_3$; thus we obtain a “simplified” precondition that has no occurrences of dt .

We note that thus obtained precondition $\bigwedge_i \bigvee_j (c_{i,j}[d/dt])[0/d] < 0$ is not necessarily equivalent to P_3 , but is stronger (i.e. less general). In our experience, however, the lost generality is most of the time marginal.

After all, the following is the (core part of the long and extensive) outcome of our prototype when it is fed with ETCS₀.

$$\begin{aligned} a_0(2\varepsilon\sqrt{2a_0(m-s-z_0)+v_0^2}+b\epsilon^2+2m-2s-2z_0) \\ + 2b\varepsilon\sqrt{2a_0(m-s-z_0)+v_0^2+a_0^2\epsilon^2+v_0^2} < 2bs \end{aligned}$$

Here v_0 and z_0 are constants that designate the initial values of v and z , respectively.

6 Implementation and Experiments

Our prototype implementation takes a WHILE^{dt} program and a postcondition as its input, and tries to calculate a precondition. The tool consists of the following components.

- A native verification condition generator implemented in OCaml. It combines the standard verification condition generation for Hoare-type logics (such as in [23]) and the previously described strategies. It relies on the *Mathematica* backend for most of the strategies as well as for (symbolic) arithmetic proofs.
- A *Mathematica* backend that provides functions for the strategies in §4–5. It also implements some proof strategies for arithmetic formulas.
- A Perl script that serves as an interface between the above two.

Our prototype employs the standard backward reasoning for Hoare-type logics. When it encounters a while loop, it tries the precondition discovery strategies in §5. If one or more of those strategies succeed and produce preconditions P_1, \dots, P_n , then our prototype continues with the disjunction $P_1 \vee \dots \vee P_n$ of those preconditions. If not, our prototype tries simplification strategies in §4. The order is *phase split, superfluous guard elimination* and then *time elapse*. If the simplification succeeds, it again tries the precondition discovery strategies. If not, our prototype reports failure.

Although we sometimes use numeric computations for finding counterexamples, the proofs are established in purely symbolic means. This is a crucial fact for the correctness guarantee via the transfer principle.

We have tested our prototype against the following WHILE^{dt} programs:

- `etcs.while`: the ETCS example used throughout the paper, and
- `zeno.while`: the behavior of a bouncing ball.

We conducted both experiments on Fujitsu HX600 with Quad Core AMD Opteron 2.3GHz CPU and 32GB memory. We used *Mathematica* 7.0 for Linux x86 (64-bit)⁶

The current implementation is premature. The goals of the experiments have been: 1) the feasibility test of our methodology of nonstandard static analysis, and 2) to drive theoretical development of static analysis strategies suited for hybrid applications. Enhancing scalability and efficiency of the prototype is left as future work.

`etcs.while`. Our prototype completed precondition generation for `etcs.while` in 40.96 seconds (31.97 seconds in the user process and 8.99 in the kernel process), generating the precondition described in §5.5 as an outcome.

`zeno.while`. We have modeled the behavior of a bouncing ball in WHILE^{dt} and run our prototype on the program. This program includes the following three parameters:

- t_0 : ending time of the system;
- b : elastic coefficient between the ground and the ball;
- h_0 : *safety height*—the ball should not reach higher than this height.

The postcondition is that the peak height of the last bounce is not higher than h_0 .

Although the current prototype does not fully succeed to compute a precondition, it manages to simplify the program substantially. For example, the ascending and descending phases in the movement of the ball are discovered automatically, using Lem. 4.2. With a manual insertion of one condition (an inequality), our prototype runs through and outputs a symbolic precondition (which is too long to present here).

Though this bouncing ball example was not fully-automatic, we still find its result interesting and encouraging. The example exhibits the *Zeno behavior*, a big challenge for many verification techniques (see e.g. [14, Section 3.3.3]). We expect our framework to be useful in Zeno-type examples, since: 1) it does not distinguish flow dynamics from jump dynamics (dynamics is always “discrete,” or “all flow dynamics is Zeno”); and 2) we do not have a governing notion of time (t is a variable just like others).

Discussion. Efficiency or speed of verification is currently not our primary concern; still we noticed several methods for speeding up. For example, calls to the four precondition discovery strategies could be parallelized. Another performance drawback of the current prototype is that the OCaml program and *Mathematica* communicate via a file written to storage, which takes time. A tighter connection via the script mode usage of *Mathematica*, or *Mathlink*, is currently looked at.

⁶ The source code of our precondition generator and the input WHILE^{dt} programs are available at <http://www-mmm.is.s.u-tokyo.ac.jp/~ichiro/papers/vcgen.tgz>

Acknowledgments. Thanks are due to all the reviewers for their careful reading and useful comments; particularly to the reviewer who pointed out an error in Lem. 5.8. I.H. is supported by Grants-in-Aid for Young Scientists (A) No. 24680001, JSPS, and by Aihara Innovative Mathematical Modelling Project, FIRST Program, JSPS/CSTP; K.S. is supported by Grants-in-Aid for JSPS Fellows 23·571.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. *Theor. Comp. Sci.* 138(1), 3–34 (1995)
2. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: EMSOFT, pp. 49–58 (2009)
3. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Non-standard semantics of hybrid systems modelers. *J. Comput. Syst. Sci.* 78(3), 877–910 (2012)
4. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 300–309. ACM (2007)
5. Bliudze, S., Krob, D.: Modelling of complex systems: Systems as dataflow machines. *Fundam. Inform.* 91(2), 251–274 (2009)
6. Chaudhuri, S., Gulwani, S., Lublinerman, R., NavidPour, S.: Proving programs robust. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT FSE, pp. 102–112. ACM (2011)
7. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear Invariant Generation Using Non-linear Constraint Solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
8. Gamboa, R.A., Kaufmann, M.: Nonstandard analysis in ACL2. *J. Autom. Reason.* 27(4), 323–351 (2001)
9. Goldblatt, R.: Lectures on the Hyperreals: An Introduction to Nonstandard Analysis. Springer (1998)
10. Gopan, D., Reps, T.: Guided Static Analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 349–365. Springer, Heidelberg (2007)
11. Hasuo, I., Suenaga, K.: Exercises in Nonstandard Static Analysis of hybrid systems. Extended version with proofs (2012), www-mmm.is.s.u-tokyo.ac.jp/~ichiro
12. Hurd, A.E., Loeb, P.A.: An Introduction to Nonstandard Real Analysis. Academic Press (1985)
13. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. *J. Log. Comput.* 20(1), 309–352 (2010)
14. Platzer, A.: Logical Analysis of Hybrid Systems—Proving Theorems for Complex Dynamics. Springer (2010)
15. Platzer, A.: The complete proof theory of hybrid systems. Tech. Rep. CMU-CS-11-144, Carnegie-Mellon Univ., Pittsburgh PA 15213 (2011)
16. Robinson, A.: Non-standard analysis, revised edn. Princeton University Press (1996)
17. Rodríguez-Carbonell, E., Tiwari, A.: Generating Polynomial Invariants for Hybrid Systems. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 590–605. Springer, Heidelberg (2005)
18. Sankaranarayanan, S.: Automatic invariant generation for hybrid systems using ideal fixed points. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 221–230. ACM (2010)
19. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: Jones, N.D., Leroy, X. (eds.) POPL, pp. 318–329. ACM (2004)

20. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. *Formal Methods in System Design* 32(1), 25–55 (2008)
21. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying Loop Invariant Generation Using Splitter Predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011)
22. Suenaga, K., Hasuo, I.: Programming with Infinitesimals: A WHILE-Language for Hybrid System Modeling. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 392–403. Springer, Heidelberg (2011)
23. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)

A Box-Based Distance between Regions for Guiding the Reachability Analysis of SpaceEx

Sergiy Bogomolov¹, Goran Frehse², Radu Grosu³, Hamed Ladan¹,
Andreas Podelski¹, and Martin Wehrle^{1,4}

¹ University of Freiburg, Germany

{bogom,ladanh,podelski}@informatik.uni-freiburg.de

² Université Joseph Fourier Grenoble 1 – Verimag, France

goran.frehse@imag.fr

³ Vienna University of Technology, Austria

radu.grosu@tuwien.ac.at

⁴ University of Basel, Switzerland

martin.wehrle@unibas.ch

Abstract. A recent technique used in falsification methods for hybrid systems relies on distance-based heuristics for guiding the search towards a goal state. The question is whether the technique can be carried over to reachability analyses that use regions as their basic data structure. In this paper, we introduce a box-based distance measure between regions. We present an algorithm that, given two regions, efficiently computes the box-based distance between them. We have implemented the algorithm in SpaceEx and use it for guiding the region-based reachability analysis of SpaceEx. We illustrate the practical potential of our approach in a case study for the navigation benchmark.

1 Introduction

The theory of hybrid systems provides a rich and popular framework for the representation of systems which incorporate both continuous and discrete behavior [2,29]. This framework has been utilized for the purpose of modeling and analyzing a large range of practically relevant systems. In particular, hybrid systems have been used for the analysis of automotive controllers [4], real-time circuits [30], and biological systems [5,1,28,35,6,18,7,21,20].

An important problem in the context of hybrid systems is the problem to determine whether a given set of bad states can be reached from an initial state. If such a set cannot be reached, the hybrid system is called safe. Unfortunately, this reachability analysis problem is decidable only for a restricted class of hybrid systems [2,22]. In order to be able to prove safety for richer classes of hybrid systems, state-of-the-art reachability tools such as SpaceEx [16] make use of over-approximations [34,10,16]. Furthermore, a lot of attention has recently been devoted to *falsification* based on testing techniques. Testing techniques are tuned to find unsafe behaviors rather than to prove safety [9,8,19,11,33,31]. In general,

falsification techniques are of great interest in industrial applications especially in the early phases of system development. The majority of these techniques are inspired from motion planning, and construct in a numeric execution-based way a rapidly exploring random tree (RRT). If the tree ends up in an unsafe state, then one has found an unsafe behavior.

In this work, we propose a best-first symbolic-reachability analysis algorithm (GBFS) which combines safety analysis and falsification. This algorithm has been added as an alternative to the current DFS algorithm of SpaceEx. GBFS produces the same result, in similar amount of time as DFS, if the bad states of the system cannot be reached from its initial states, despite SpaceEx inherent over-approximations. However, GBFS is much faster than DFS on our benchmarks in producing a symbolic counterexample if the system is potentially unsafe. The heuristic used to guide the search is based on an appropriate cost measure for hybrid systems based on dwell times.

For a given state s , the cost measure estimates the search effort to reach an error state from s . The search preferably explores states with smaller estimated costs, and thus avoids exploring unnecessary states. Obviously, to obtain an overall efficient model checking approach, cost measures are both supposed to guide the search accurately and to be efficiently computable. Guided search has recently been successfully applied in the context of discrete and timed systems [13,32,25,24,27,12,36,26]. In those contexts, the costs of a state s have been defined as the smallest number of transitions in the state space to reach a nearest error state from s . Overall, guided search has shown to be able to significantly improve the efficiency of model checking for these classes of systems.

While measuring the costs of states in terms of transitions in the state space is an appropriate method for discrete and timed systems, the situation becomes slightly more complex for the more general class of hybrid systems. In the context of hybrid systems, the overall model checking time specifically depends on the operations to compute the continuous post of states because this operation is most expensive during the exploration of the state space. Moreover, in contrast to discrete and timed systems, the costs of the post operation depend on the *dwell time*, i. e., the amount of time spent in the corresponding location. This dependence occurs, at least in practice, because common tools like SpaceEx compute the post operation based on iteratively computing the continuous image of the region in intermediate points. Therefore, in the context of hybrid systems, it is desirable to explore traces with low accumulated dwell time.

We theoretically show that, for a certain class of hybrid systems, our cost measure provides desired search behavior. Moreover, although hard to compute exactly, the representation of our cost measure lends itself to an accurate (box-based) cost measure as an approximation. This approximation turns out to be efficiently computable and, although surprisingly simple, to accurately guide the search in the state space. Our experiments with SpaceEx show the practical potential on challenging benchmarks.

The paper is organized as follows. In Sec. 2 we introduce the preliminaries for the paper. Sec. 3 introduces the main contribution of this work based on a

trajectory-based cost measure. This cost measure is experimentally evaluated in Sec. 4. Finally, we conclude the paper in Sec. 5.

2 Preliminaries

In this section, we give the preliminaries for this paper. In Sec. 2.1, we introduce our computational model. In Sec. 2.2, we present a basic reachability algorithm for the state space exploration. Based on the reachability algorithm, guided search and cost measures are introduced in Sec. 2.3.

2.1 Notation

In this paper, we consider models that can be represented by hybrid systems.

Definition 1 (Hybrid System). A hybrid system is formally a tuple $\mathcal{H} = (\text{Loc}, \text{Var}, \text{Init}, \text{Flow}, \text{Trans}, \text{Inv})$ defining

- the finite set of locations Loc ,
- the set of continuous variables $\text{Var} = \{x_1, \dots, x_n\}$ from \mathbb{R}^n ,
- the initial condition, given by the constraint $\text{Init}(\ell)$ for each location ℓ ,
- the continuous transition relation, given by the expression $\text{Flow}(\ell)(v)$ for each continuous variable v and each location ℓ . We assume $\text{Flow}(\ell)$ to be of the form

$$\dot{x}(t) = Ax(t) + u(t), u(t) \in \mathcal{U},$$

where $x(t) \in \mathbb{R}^n$, A is a real-valued $n \times n$ matrix and $\mathcal{U} \subseteq \mathbb{R}^n$ is a closed and bounded convex set.

- the discrete transition relation, given by a set Trans of discrete transitions; a discrete transition is formally a tuple (ℓ, g, ξ, ℓ') defining
 - the source location ℓ and the target location ℓ' ,
 - the guard, given by a linear constraint g ,
 - the update, given by a affine mapping ξ ,
- the invariant, given by the linear constraint $\text{Inv}(\ell)$ for each location ℓ .

A state of the hybrid system \mathcal{H} is a tuple (ℓ, \mathbf{x}) consisting of a location $\ell \in \text{Loc}$ and a point $\mathbf{x} \in \mathbb{R}^n$, i.e., \mathbf{x} is an evaluation of the continuous variables in Var .

The semantics of a hybrid system is defined in terms of its *trajectories*. Let $\mathcal{T} = [0, \Delta]$ for $\Delta \geq 0$. A trajectory of a hybrid system \mathcal{H} from state $s = (\ell, \mathbf{x})$ to state $s' = (\ell', \mathbf{x}')$ is defined by a tuple $\rho = (L, \mathbf{X})$, where $L : \mathcal{T} \rightarrow \text{Loc}$ and $\mathbf{X} : \mathcal{T} \rightarrow \mathbb{R}^n$ are functions that define for each time point in \mathcal{T} the location and values of the continuous variables, respectively. For a given trajectory ρ , we define a sequence of time points where location switches happen by $(\tau_i)_{i=0 \dots k} \in \mathcal{T}^{k+1}$. In such case we say that the trajectory ρ has discrete length $|\tau| = k$. Trajectories $\rho = (L, \mathbf{X})$ (and the corresponding sequence $(\tau_i)_{i=0 \dots k}$) have to satisfy the following conditions:

- $\tau_0 = 0$, $\tau_i < \tau_{i+1}$, and $\tau_k = \Delta$ – the sequence of switching points increases, starts with 0 and ends with Δ

- $L(0) = \ell$, $\mathbf{X}(0) = \mathbf{x}$, $L(\Delta) = \ell'$, $\mathbf{X}(\Delta) = \mathbf{x}'$ – the trajectory starts in $s = (\ell, \mathbf{x})$ and ends in $s' = (\ell', \mathbf{x}')$
- $\forall i \forall t \in [\tau_i, \tau_{i+1}] : L(t) = L(\tau_i)$ – the location is not changed during the continuous evolution
- $\forall i \forall t \in [\tau_i, \tau_{i+1}] : \mathbf{X}(t) \in Flow(L(\tau_i))$, i.e. $\dot{\mathbf{X}}(t) = A\mathbf{X}(t) + u(t)$ holds and thus the continuous evolution is consistent with the differential equations of the corresponding location
- $\forall i \forall t \in [\tau_i, \tau_{i+1}] : \mathbf{X}(t) \in Inv(L(\tau_i))$ – the continuous evolution is consistent with the corresponding invariants
- $\forall i \exists (L(\tau_i), g, \xi, L(\tau_{i+1})) \in Trans : \mathbf{X}_{end}(i) = \lim_{\tau \rightarrow \tau_{i+1}^-} \mathbf{X}(\tau) \wedge \mathbf{X}_{end}(i) \in g \wedge \mathbf{X}(\tau_{i+1}) = \xi(\mathbf{X}_{end}(i))$ – every continuous transition is followed by a discrete one, $\mathbf{X}_{end}(i)$ defines the values of continuous variables right before the discrete transition at the time moment τ_{i+1} whereas $\mathbf{X}_{start}(i) = \mathbf{X}(\tau_i)$ denotes the values of continuous variables right after the switch at the time moment τ_i .

We say that s' is *reachable* from s if a trajectory from s to s' exists. The reachable state space $\mathcal{R}(\mathcal{H})$ of \mathcal{H} is defined as the set of states such that a state s is contained in $\mathcal{R}(\mathcal{H})$ iff s is reachable from S_{init} . In this paper, we also refer to *symbolic states*. A symbolic state $s = (\ell, R)$ is defined as a tuple, where $\ell \in Loc$, and R is a convex and bounded set consisting of points $\mathbf{x} \in \mathbb{R}^n$. The symbolic part of a symbolic state is also called *region*. The symbolic state space of \mathcal{H} is called the *region space*. The initial set of states S_{init} of \mathcal{H} is defined as $\bigcup_\ell (\ell, Init(\ell))$.

In this paper, we assume there is a given set of *error states* that violate a given property. Our goal is to find a trajectory from S_{init} to an error state. A trajectory that starts in a state s and leads to an error state is called an *error trajectory* $\rho_e(s)$. As already outlined in the introduction, the time to compute an error trajectory $\rho_e(s)$ can significantly depend on the accumulated *dwell times* of the locations in $\rho_e(s)$. Therefore, we define the costs of a state s as

$$cost(s) := \min_{\rho_e(s)} \sum_{i=0}^{|\tau|-1} \delta_i,$$

i.e., as the minimal sum of dwell times ranged over the error traces that start in s , where $\delta_i = \tau_{i+1} - \tau_i$. Error trajectories can be found by searching in the state space. We will give a short introduction to this search-based approach in the next section.

2.2 Finding Error States through Search

We introduce a standard basic reachability algorithm along the lines of the reachability algorithm used by SpaceEx. It works on the region space of a given hybrid system. The algorithm checks if a symbolic error state in a given set S_{error} is reachable from a given set of symbolic initial states S_{init} . We define a symbolic state s in the region space of \mathcal{H} to be an error state if there is a symbolic state

Algorithm 1. Basic reachability algorithm

Input: Set of initial symbolic states S_{init} , set of error states S_{error}
Output: Can a symbolic state in S_{error} be reached from a symbolic state in S_{init} ?

```

1: PUSH ( $\mathcal{L}_{passed}, S_{init}$ )
2: PUSH ( $\mathcal{L}_{waiting}, S_{init}$ )
3:  $i \leftarrow 0$ 
4: while ( $\mathcal{L}_{waiting} \neq \emptyset \wedge i < i_{max}$ ) do
5:    $s_{curr} = \text{GETNEXT } (\mathcal{L}_{waiting})$ 
6:    $i \leftarrow i + 1$ 
7:   if  $s_{curr} \in S_{error}$  then
8:     return "Error state reached"
9:   end if
10:   $S' \leftarrow \text{COMPUTESUCCESSORS}(s_{curr})$ 
11:  for all  $s' \in S'$  do
12:    if  $s' \notin \mathcal{L}_{passed}$  then
13:      PUSH ( $\mathcal{L}_{passed}, s'$ )
14:      PUSH ( $\mathcal{L}_{waiting}, s'$ )
15:    end if
16:  end for
17: end while
18: return "Error state not reachable"
```

$s_e \in S_{error}$ such that s and s_e agree on their discrete part, and the intersection of the regions of s and s_e is not empty.

Starting with the set of initial symbolic states from S_{init} , the algorithm explores the region space of a given hybrid system by iteratively computing symbolic successor states until an error state is found, no more states remain to be considered, or a (given) maximum number of iterations i_{max} is reached.

In the following, we explain Alg. II in more detail. Symbolic states for which the successor states have been computed are called *explored*, whereas symbolic states that have been computed but not yet explored are called *visited*. Both visited and explored states are stored in a dedicated data structure \mathcal{L}_{passed} . Symbolic states in \mathcal{L}_{passed} are used to detect cycles in the region space (see below). Moreover, there is a data structure $\mathcal{L}_{waiting}$ that contains visited states that have not necessarily been explored yet. An iteration of the algorithm consists of several steps. First, a symbolic state s is taken from $\mathcal{L}_{waiting}$ and checked if s is an error state. If this is the case, the algorithm terminates. If s is not an error state, the symbolic successor states of s are computed (which in turn is a 2-step operation consisting of the computation of the discrete and continuous post state; we omit a more detailed description here). To avoid exploring cycles in the region space, symbolic successor states that are already contained in \mathcal{L}_{passed} are not considered again; the others are stored in $\mathcal{L}_{waiting}$.

The order in which the region space is explored by Alg. II depends on the implementation of $\mathcal{L}_{waiting}$ (e.g., a queue-based implementation corresponds to breadth-first search). Specifically, *guided search* preferably explores states that

appear to be more promising according to a cost measure. We will describe this approach in more detail in the next section.

2.3 Guided Search

Guided search is an instantiation of the basic reachability algorithm that has been introduced in the previous section. As a first characteristic of guided search algorithms, $\mathcal{L}_{\text{waiting}}$ is implemented as a priority queue. Therefore, the PUSH function additionally requires a priority (cost) value for the pushed state, and the GETNEXT function (line 5 in Alg. 1) returns a state with best priority according to the cost measure. In the following, we discuss a desirable property of cost measures in the context of guided search. As already outlined, we intend to design a cost measure that guides the search well in the region space. To achieve good guidance, the relative error of a cost measure h to the cost function as defined in the previous section is not necessarily correlated to the accuracy of h . In other words, h may accurately guide the search although the relative error of h 's cost estimations is high. This is because it suffices for h to always select the “right” state to be explored next.¹ Based on this observation, we give the definition of *order-preserving*.

Definition 2 (Order-Preserving). Let \mathcal{H} be a hybrid system. A cost measure h is order-preserving if for all states s and s' with $\text{cost}(s) < \text{cost}(s')$, then also $h(s) < h(s')$.

Cost measures that are order-preserving lead to perfect search behavior with respect to the cost function. Therefore, it is desirable to have cost measures that satisfy this property. We will come back to this point in the next section.

3 The Box-Based Distance Measure

In this section, we present the main contribution of this work. In Sec. 3.1, we provide a conceptual description of an idealized distance measure based on the length of trajectories. This idealized distance measure is used as the basis for our box-based distance measure which is presented in Sec. 3.2.

3.1 A Trajectory-Based Distance Measure

In this section, we formulate a distance measure dist that can be expressed in terms of the length of trajectories (see below for a justification of the name). For states s and s' , the distance measure $\text{dist}(s, s')$ is defined as the minimal length of a trajectory ρ that is obtained from the continuous flow and discrete switches

¹ As a simple example, consider two states s and s' with real costs 100 and 200, respectively. Furthermore, consider a cost measure that estimates the costs of these states as 1 and 2, respectively. We observe that the relative error is high, but the better state is determined nevertheless.

of trajectories that lead from s to s' . To define this more formally, we denote the set of trajectories that lead from s to s' with $\mathcal{T}(s, s')$. Moreover, $dist_{eq}(\mathbf{x}, \mathbf{x}')$ denotes the Euclidean distance between points \mathbf{x} and \mathbf{x}' . Using this notation, we give the definition of our trajectory-based distance measure.

Definition 3 (Trajectory-Based Distance Measure). Let \mathcal{H} be a hybrid system, let s and s' be states of \mathcal{H} . We define the distance measure

$$dist(s, s') := \min_{\rho \in \mathcal{T}(s, s')} \sum_{i=0}^{|\tau|-1} \left(\int_{\tau_i}^{\tau_{i+1}} \sqrt{\dot{x}_1^2(t) + \dots + \dot{x}_n^2(t)} dt + dist_{eq}(i, i+1) \right),$$

where $\rho = (L, \mathbf{X})$, $\mathbf{X}(t) = (x_1(t), \dots, x_n(t))$, and $dist_{eq}(i, i+1)$ is a short-hand for $dist_{eq}(\mathbf{X}_{end}(i), \mathbf{X}_{start}(i+1))$.

Informally speaking, the distance between states s and s' is defined as the length of a shortest trajectory ρ from s to s' induced by the differential equations and discrete updates of the visited locations $L(\tau_i)$ in ρ . Obviously, the trajectory-based distance measure can be applied to error states in a straight-forward way by setting s' to an error state. We call the trajectory-based error distance measure $dist_E(s) := \min_{s_e} dist(s, s_e)$, where s_e ranges over the set of given error states of \mathcal{H} .

In the following, we show that for a certain class of hybrid systems \mathcal{H} , $dist(s)$ is indeed correlated to the costs of s for all states s of \mathcal{H} . In fact, this correlation can be established for hybrid systems such that

1. all differential equations in \mathcal{H} are of the form $\dot{x}_i(t) = \pm c_i$ for every continuous variable $x_i \in Var$ and a constant $c_i \in \mathbb{N}$, and
2. all guards in \mathcal{H} do not contain discrete updates.

We call hybrid systems that satisfy the above requirements *restricted systems*. Specifically, we observe that a necessary condition for hybrid system \mathcal{H} to be a restricted system is that for every continuous variable x_i in \mathcal{H} , there is a global constant $c_i \in \mathbb{N}$ such that *all* differential equations in \mathcal{H} that talk about x_i only differ in the sign. It is not difficult to see that for the class of restricted systems, the length of the obtained flow is linearly correlated with the time. Therefore, the error distance measure $dist_E$ is order-preserving.

Proposition 1. For restricted systems \mathcal{H} , $dist_E$ is order-preserving.

Proof. We show that from $cost(s) < cost(s')$, it follows that $dist_E(s) < dist_E(s')$. As \mathcal{H} is a restricted system, the square root of $\dot{x}_1^2(t) + \dots + \dot{x}_n^2(t)$ is constant and $dist_{eq}(i, i+1)$ is equal to zero. Therefore, $dist_E(s) = \min_{s_e} \min_{\rho \in \mathcal{T}(s, s_e)} c \sum \delta_i$, which is equal to $c \cdot cost(s)$. This proves the claim.

Prop. II leads to an interesting and important observation. Roughly speaking, we have reduced the problem of computing (dwell time) costs in the state space to the problem of computing “shortest” flows between regions. Therefore, Prop. II shows that under certain circumstances, we can choose between *cost* and *dist* without loosing precision. However, although still hard to compute, the representation of *dist* based on lengths of flows lends itself to an approximation based on *estimated* flow lengths. This approximation is presented in the next section.

3.2 The Box-Based Approximation

In the following, we propose an effective approximation of the $dist$ function that we have derived in the last section. While the $dist$ measure has been defined for concrete states, our box-based approximation is defined for symbolic states. The approximation is based on the following two ingredients.

1. Instead of computing the exact length of trajectories between two points \mathbf{x} and \mathbf{x}' (as required in Def. 3), we use the Euclidean distance between \mathbf{x} and \mathbf{x}' .
2. As we are working in the region space, we approximate a given region R with the smallest box \mathcal{B} such that R is contained in \mathcal{B} . This corresponds to the well-known principle of Cartesian abstraction.

In the following, we will discuss these ideas and make them precise. As stated, we define the estimated distance between points \mathbf{x} and \mathbf{x}' as the Euclidean distance between \mathbf{x} and \mathbf{x}' . Unfortunately, the Euclidean distance is not order-preserving for restricted systems, but only for even more restricted systems that allow even less behavior. This is formalized in the following proposition. For a state $s = (\ell, \mathbf{x})$, we define $dist_E^{eq}(s) := \min_{s_e} dist_{eq}(\mathbf{x}, \mathbf{x}_e)$, where $s_e = (\ell_e, \mathbf{x}_e)$ ranges over the error states, and $dist_{eq}$ is the Euclidean distance function as introduced earlier.

Proposition 2. *For restricted systems \mathcal{H} with $\dot{x}_i(t) = c_i$, i.e., for restricted systems where all locations have the same continuous behavior, $dist_E^{eq}$ is order-preserving.*

Proof. We show that from $cost(s) < cost(s')$, it follows that $dist_E^{eq}(s) < dist_E^{eq}(s')$. By assumption, \mathcal{H} is a restricted system where every location has the same continuous dynamics. Therefore, the Euclidean distance $dist_{eq}(s, s_e)$ is equal to $\int_0^{\tau_k} \sqrt{\dot{x}_1^2(t) + \dots + \dot{x}_n^2(t)} dt$, where τ_k is equal to the accumulated dwell time of the trajectory from s to s_e . Furthermore, the square root of $\dot{x}_1^2(t) + \dots + \dot{x}_n^2(t)$ is some constant c . Thus $dist_E^{eq}(s) = \min_{s_e} dist_{eq}(s, s_e) = \min_{s_e} c \cdot \tau_k = c \cdot \min_{s_e} \tau_k$ which in turn is equal to $c \cdot cost(s)$.

The above proposition reflects that the Euclidean distance is a coarse approximation of the trajectory-based distance measure because it is effectively only order-preserving for systems that allow behavior that corresponds to systems with only one location. Indeed, it is the coarsest approximation one can think of on the one hand. However, on the other hand, we have shown that there *exist* systems for which it *is* order-preserving, which suggests (together with Prop. 1) that the Euclidean distance could be a good heuristic to estimate distances also for richer classes of hybrid systems. Moreover, it is efficiently computable which is particularly important for distance heuristics that are computed on-the-fly during the state space exploration. (Obviously, one can think of arbitrary more precise approximations based on piecewise linear functions; however, such approximations also become more expensive to compute. We will come back to this point in the conclusions.)

For our distance heuristic, we approximate a given symbolic state $s = (\ell, R)$ with the smallest box $\mathcal{B}(s)$ that contains R . Formally, this corresponds to the requirement

$$R \subseteq \mathcal{B}(s) = [x_1, x'_1] \times \dots [x_n, x'_n] \subseteq \mathbb{R}^n \wedge \forall \mathcal{B}' \neq \mathcal{B}(s) : R \subseteq \mathcal{B}' \Rightarrow \mathcal{B}(s) \subseteq \mathcal{B}'.$$

In order to be efficiently computable, it is essential that tight over-approximating boxes can be computed efficiently. This can be achieved using linear programming techniques. Our distance heuristic h^{eq} is defined as the Euclidean distance between the center of two boxes. Formally, for a symbolic state $s = (\ell, R)$, we define

$$h^{eq}(s) := \min_{s_e} dist_{eq}(\text{CENTER}(\mathcal{B}(R)), \text{CENTER}(\mathcal{B}(R_e))),$$

where $s_e = (\ell_e, R_e)$ ranges over the set of error states of \mathcal{H} , $dist_{eq}$ is the Euclidean distance metric, and $\text{CENTER}(B)$ denotes the central point of box B . Obviously, central points of boxes can be computed efficiently as the arithmetic average of its lower and upper bounds for every dimension.

Overall, our distance heuristic h^{eq} determines distance estimations for symbolic states $s = (\ell, R)$ by first over-approximating R with the smallest box \mathcal{B} that contains R , and then computing the minimal Euclidean distance between \mathcal{B} 's center and the center of an error state. This procedure is summed up by Alg. 2.

Algorithm 2. COMPUTE DISTANCE HEURISTIC h^{eq}

Input: State $s = (\ell, R)$

Output: Estimated distance to a closest error state in S_{error}

```

1:  $d_{min} \leftarrow \infty$ 
2:  $\mathcal{B} \leftarrow \mathcal{B}(R)$ 
3: for all  $s' = (\ell', R') \in S_{error}$  do
4:    $\mathcal{B}' \leftarrow \mathcal{B}(R')$ 
5:    $d_{curr} \leftarrow dist_{eq}(\text{CENTER}(\mathcal{B}), \text{CENTER}(\mathcal{B}'))$ 
6:   if  $d_{curr} < d_{min}$  then
7:      $d_{min} \leftarrow d_{curr}$ 
8:   end if
9: end for
10: return  $d_{min}$ 
```

4 Experiments

We have implemented our box-based distance heuristic h^{eq} in SpaceEx and compare the resulting guided search algorithm to the standard depth-first search (DFS) algorithm of SpaceEx. The experiments have been performed on a machine with an Intel Core i3 2.40GHz processor and with 4 GB of memory. For both search settings (i.e., for guided search as well as for uninformed DFS), SpaceEx has been run with the same parameters (see Sec. 4.2).

In the following, we first introduce our benchmark problems in Sec. 4.1. Afterwards, the experimental results are presented and discussed in Sec. 4.2.

4.1 Case Studies

For the evaluation of our approach, we used several problem instances of the two case studies *Navigation benchmark* and *System of Tanks*.

Navigation Benchmark. As a first case study, we apply the *navigation benchmark* that has been proposed in the literature [14]. In the scope of this benchmark, we consider an object moving in the plane. The plane is divided into the grid of squares where some initial state is given (i.e., region, velocity in direction x and velocity in direction y). Furthermore, for each square, some differential equations are defined which govern the system in the considered square. Finally, some square A which should be reached and some square B which is to be avoided are defined. We will look for a path from the initial state to square A . We consider different problem instances of this benchmark with different sizes of the grid.

System of Tanks. As a second benchmark which is similar to the one presented by Frehse and Maler [17], we consider a network of tanks which are connected by channels. Tanks have a fixed capacity, channels are characterized by their throughputs. We have several kinds of tanks with different functionality, namely *production*, *buffer*, *delivery buffer*, and *reactor*. The functionality of these tanks is as follows. Liquid can be delivered from the production to one of the buffers B_1 , B_2 or B_3 . In particular, liquid can be stored in those buffers for some time. Furthermore, liquid is forwarded to reactors R_1 , R_2 and R_3 . Liquid can be transferred only from B_i to R_i . Finally, liquid is transferred from the reactors to the delivery buffer B_D from which it is directly delivered to the customer.

For a given control strategy, it must be ensured that the delivery buffer B_D never gets empty according to this strategy (i.e., requirements of the customer are satisfied). For a fixed $i \in \{1, 2, 3\}$, we have equal throughput v_i from production to B_i , from B_i to R_i , and from R_i to B_D . Finally, the transmission rate from B_D to the customer is defined by v_{out} .

We will investigate the behavior of the following controller. The controller has the following phases which are characterized by constant m defining the length of the time period when liquid is transferred between tanks:

1. $time = 0$: $open(Production, B_i)$ - transfer of liquid from *Production* to B_i starts
2. $time \in [0, m]$: $Production \rightarrow B_i$ - buffer B_i is filled
3. $time = m$: $close(Production, B_i)$ - stop filling B_i
4. $time = m$: $open(B_i, R_i)$ - start transfer to R_i
5. $time \in [m, 2m]$: $B_i \rightarrow R_i$ - liquid is transferred from the buffer B_i to the reactor R_i

Table 1. Experimental results of SpaceEx for the navigation benchmark with uninformed depth-first search and guided search. Abbreviations: DFS: depth-first search, DTime: overall accumulated dwell time for all explored states, time in s: overall search time of SpaceEx in seconds.

Benchmark instance	DFS			Guided search		
	#iterations	DTime	time in s	#iterations	DTime	time in s
NAV25	200	245.7	160.157	43	111.1	44.891
NAV26	200	391.5	327.66	44	110.2	57.95
NAV27	200	539.6	366.621	49	121.4	59.212
NAV28	47	96.3	63.176	34	99.8	50.528
NAV29	162	410.5	217.521	42	133.1	66.479
NAV30	174	308.6	176.457	40	129.4	69.779

6. $time = 2m$: $close(B_i, R_i)$ - stop transferring to R_i
7. $time = 2m$: $open(R_i, B_D)$ - start transfer to B_D
8. $2m$ sec - $3m$ sec: $R_i \rightarrow B_D$ - liquid is transferred from the reactor R_i to the delivery buffer B_D , i.e. delivery buffer is refilled.
9. $time = 3m$: $close(R_i, B_D)$ - stop refilling B_D . After this phase the controller goes back to the phase 1.

The controller can non-deterministically choose which buffer (and reactor) to use in each iteration. Thus the number of trajectories grows exponentially with time. We assume the consumption to be constant. Thus it is essential to transfer to the delivery buffer enough liquid so it does not get empty within phases in which the level of liquid in the delivery buffer sinks.

Our goal is to check whether the presented controlled may fail, i.e., may lead to the drainage of the production buffer, and to discover the possible failure as soon as possible. For our test, we set v_1, v_2, v_3 such that $v_3 > v_1 > v_2$, and v_{out} such that the delivery buffer will get empty no matter which buffer is chosen by the controller. However, choosing B_2 with the smallest throughput (and thus refillment) rate v_2 will obviously lead to the faster drainage of the delivery buffer and therefore to the faster discovery of the controller failure. Contrarily, the unfortunate choice of v_3 will lead to the delay in the controller failure discovery.

4.2 Experimental Results

The experimental results for the largest problem instances in the navigation benchmark (NAV25, ..., NAV30) are provided in Table 1. The results have been obtained using the LGG support function scenario of SpaceEx. Template polyhedra are represented using 32 uniform directions. Furthermore, the maximal number of iterations is set to 200, and the continuous sampling time is set to 0.1 seconds. Finally, the local time horizon for the continuous post operation is set to 40. We compare the number of iterations, the search time, as well as the overall accumulated dwell time during the exploration of the state space of SpaceEx. The accumulated dwell time serves as an additional measure

to compare the “quality” of the search guidance because, as argued in the previous sections, this time is correlated with overall search effort, and our distance heuristic tries to minimize it.

First, we observe that in all these problem instances, the guided search algorithm with our box-based distance heuristic could significantly improve the overall performance of the model checking process. Specifically, we observe that the overall accumulated dwell time is reduced when guiding the search, which apparently results in a lower search effort. Furthermore, the number of iterations of SpaceEx reduces. As a side remark, for NAV25, NAV26 and NAV27, the standard depth-first search did not find a solution after the maximal number of 200 iterations.

Let us consider the results for the largest navigation benchmark problem, NAV30, in more detail. Fig. 1 and Fig. 2 graphically compare the way of the object in NAV30 while moving over the 25×25 grid and searching for the target state. The initial region of the object is on the left above, the goal region is on the right below. We observe that, using depth-first-search as shown in Fig. 1, the object reaches the target state on a trace with a considerable (circle-shaped) detour on the one hand. On the other hand, using guided search with the h^{eq} distance heuristic as shown in Fig. 2, the way of the object apparently becomes more straight. As a consequence, with uninformed depth-first search, SpaceEx needs 174 iterations and over 176 seconds to reach the target state. In contrast, with guided search, SpaceEx finds the target state within only 40 iterations, resulting in a remarkable speed-up considering the overall search time.

Considering the experimental results for the benchmark problems based on the system of tanks, we first report that our h^{eq} distance heuristic does not provide further guidance information for the state space exploration in the classical search setting. More precisely, for a symbolic state s of that system, all successor states of s are equally evaluated by h^{eq} . This shows that, unsurprisingly,

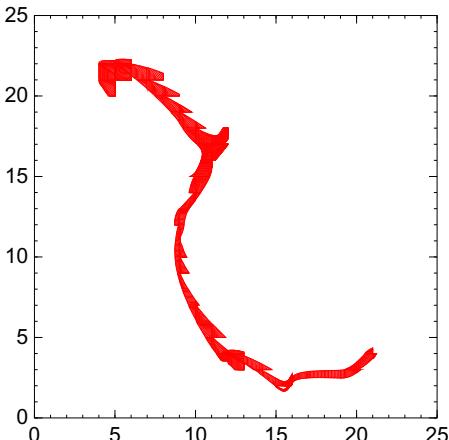


Fig. 1. Depth-first search

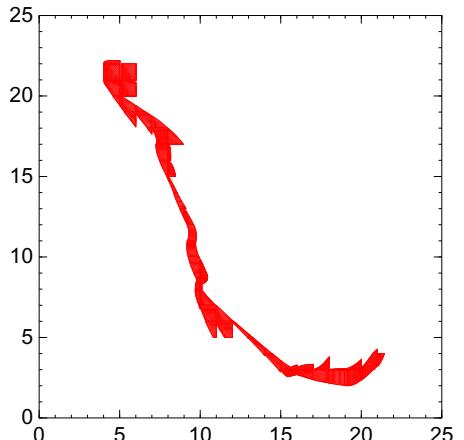


Fig. 2. Guided search with h^{eq}

Table 2. Experimental results of SpaceEx for the tank benchmark with uninformed depth-first search and guided search with look-ahead. Abbreviations: DFS: depth-first search, DTime: overall accumulated dwell time for all explored states, time in s: overall search time of SpaceEx in seconds.

Benchmark instance	DFS			Guided search		
	#iterations	DTime	time in s	#iterations	DTime	time in s
TANK01	1396	34900	7.356	207	3850	3.962
TANK02	3712	92800	42.179	526	9850	8.952
TANK03	6034	150850	105.22	846	15850	14.876
TANK04	8349	208750	196.69	1166	21850	21.311

h^{eq} does not fire appropriately in all problem domains. However, h^{eq} can still give benefits compared to uninformed depth-first search in the context of *look-aheads* in the state space. For a given state s , a look-ahead works by not only considering heuristic values of the direct successor states of s , but also by considering heuristic values of successor states in a fixed depth greater than one. In Table 2, we report experimental results within this search setting. We have used the PHAVer scenario of SpaceEx. In addition, the maximal number of iterations is set to 250000, and the continuous sampling time is set to 1 second. Finally, the local time horizon for the continuous post operation is set to 200. The problems TANK01, ..., TANK04 are benchmark instances of increasing complexity that differ in the initial level of liquid in the delivery buffer.

Considering these results, we again observe that guided search can significantly outperform uninformed depth-first search of SpaceEx. In particular, the number of iterations and the overall search time could be considerably reduced.

We conclude the section with a short discussion for which kind of systems our approach is suited best. First, we observe that our heuristic is especially accurate when the system dynamics depends only on the continuous state. In particular, this is the case for the navigation benchmark. In general, systems of that kind occur in practice when complex dynamics is approximated with a simpler one through state space partitioning: For example, the phase portrait approximations [23], the approximation techniques employed by PHAVer [15] and hybridization techniques [3] fall into this category. Additionally considering look-aheads (i. e., considering not only heuristic values of direct successor states, but also of states in a fixed depth greater than one) is useful when crucial changes of a system state may arise after performing *several* steps (as it is the case in the system of tanks).

5 Conclusions

In this paper we have introduced a best-first symbolic-reachability analysis algorithm (GBFS) for a particular class of hybrid systems, the ones that have a linear behavior (in the control-theoretic sense) in each mode. The algorithm has been added as an additional reachability-analysis engine to SpaceEx, the state-of-the-art reachability-analysis tool for this class of systems [16].

GBFS takes advantage of the symbolic-computation routines of SpaceEx, and in particular of its efficient computation of the smallest box enclosing a symbolic region. As a consequence, the algorithm has a similar time-complexity for reachability analysis as the depth-first search algorithm (DFS) of this tool.

GBFS is tuned for efficient falsification, where it considerably outperforms DFS on our benchmarks. The improved efficiency is achieved by choosing the successor region which has the smallest Euclidean distance between the center of its enclosing box and the center of the box enclosing the bad-region. We have shown that for a particular class of hybrid systems, this metric is an appropriate approximation of an idealized trajectory metric. Our experimental evaluation additionally shows that this metric can serve as an informed cost heuristic even for richer classes of hybrid systems.

For the future, it will be interesting to further refine our box-based distance metric. In this paper, we have chosen the most canonical way as an approximation; however, we also have already outlined that arbitrary more precise approximations based on piecewise linear functions are possible. In this context, an important topic for future research will be the question how much precision can be gained while still being efficiently computable.

Acknowledgments. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS, <http://www.avacs.org/>).

References

1. Alur, R., Belta, C., Ivančić, F., Kumar, V., Mintz, M., Pappas, G.J., Rubin, H., Schug, J.: Hybrid Modeling and Simulation of Biomolecular Networks. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 19–32. Springer, Heidelberg (2001)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicolin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138, 3–34 (1995)
3. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of non-linear systems. *Acta Informatica* 43(7), 451–476 (2007)
4. Balluchi, A., Benvenuti, L., Di Benedetto, M.D., Pinello, C., Sangiovanni-Vincentelli, A.L.: Automotive engine control and hybrid systems: challenges and opportunities. *Proceedings of the IEEE* 88(7), 888–912 (2000)
5. Barbano, P., Spivak, M., Feng, J., Antoniotti, M., Misra, B.: A coherent framework for multi-resolution analysis of biological networks with memory: Ras pathway, cell cycle and immune system. *National Academy of Science*, 6245–6250 (2005)
6. Batt, G., Belta, C., Weiss, R.: Model Checking Genetic Regulatory Networks with Parameter Uncertainty. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 61–75. Springer, Heidelberg (2007)
7. Belta, C., Finin, P., Habets, L.C.G.J.M., Halász, Á.M., Imieliński, M., Kumar, R.V., Rubin, H.: Understanding the Bacterial Stringent Response Using Reachability Analysis of Hybrid Systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 111–125. Springer, Heidelberg (2004)

8. Bhatia, A., Frazzoli, E.: Incremental Search Methods for Reachability Analysis of Continuous and Hybrid Systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 142–156. Springer, Heidelberg (2004)
9. Branicky, M.S., Curtiss, M.M.: Nonlinear and hybrid control via RRTs. In: Symp. on Mathematical Theory of Networks and Systems (2002)
10. Chutinan, C., Krogh, B.H.: Computational techniques for hybrid system verification. *IEEE Transactions on Automatic Control* 48(1), 64–75 (2003)
11. 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)
12. Dräger, K., Finkbeiner, B., Podelski, A.: Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer* 11(1), 27–37 (2009)
13. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer* 5(2), 247–267 (2004)
14. Fehnker, A., Ivančić, F.: Benchmarks for Hybrid Systems Verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004)
15. Frehse, G.: PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
16. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
17. Frehse, G., Maler, O.: Reachability Analysis of a Switched Buffer Network. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 698–701. Springer, Heidelberg (2007)
18. Ghosh, R., Tomlin, C.J.: Symbolic reachable set computation of piecewise affine hybrid automata and its application to biological modeling: Delta-notch protein signaling. *IEEE Transactions on Systems Biology* 1(1), 170–183 (2004)
19. Girard, A., Pappas, G.J.: Verification Using Simulation. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 272–286. Springer, Heidelberg (2006)
20. Grosu, R., Batt, G., Fenton, F.H., Glimm, J., Le Guernic, C., Smolka, S.A., Bartocci, E.: From Cardiac Cells to Genetic Regulatory Networks. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 396–411. Springer, Heidelberg (2011)
21. Grosu, R., Smolka, S.A., Corradini, F., Wasilewska, A., Entcheva, E., Bartocci, E.: Learning and detecting emergent behavior in networks of cardiac myocytes. *Communications of the ACM (CACM)* 52(3), 1–10 (2009)
22. Henzinger, T., Kopke, P., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: ACM Symposium on Theory of Computing, pp. 373–382 (1995)
23. Henzinger, T., Wong-Toi, H.: Linear Phase-Portrait Approximations for Nonlinear Hybrid Systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 377–388. Springer, Heidelberg (1996)
24. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: UPPAAL/DMC – Abstraction-Based Heuristics for Directed Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 679–682. Springer, Heidelberg (2007)

25. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrman, G.: Adapting an AI Planning Heuristic for Directed Model Checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 35–52. Springer, Heidelberg (2006)
26. Kupferschmid, S., Wehrle, M.: Abstractions and Pattern Databases: The Quest for Succinctness and Accuracy. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 276–290. Springer, Heidelberg (2011)
27. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster Than UPPAAL? In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 552–555. Springer, Heidelberg (2008)
28. Lincoln, P., Tiwari, A.: Symbolic Systems Biology: Hybrid Modeling and Analysis of Biological Networks. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 660–672. Springer, Heidelberg (2004)
29. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata. Inf. and Comp. 185(1), 103–157 (2003)
30. Maler, O., Yovine, S.: Hardware timing verification using kronos. In: Israeli Conference on Computer Systems and Software Engineering (1996)
31. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Hybrid Systems: From Verification to Falsification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 463–476. Springer, Heidelberg (2007)
32. Qian, K., Nyman, A.: Guided Invariant Model Checking Based on Abstraction and Symbolic Pattern Databases. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 497–511. Springer, Heidelberg (2004)
33. Ratschan, S., Smaus, J.G.: Verification-Integrated falsification of Non-Deterministic hybrid systems. In: Analysis and Design of Hybrid Systems (2006)
34. Silva, B., Stursberg, O., Krogh, B., Engell, S.: An assessment of the current status of algorithmic approaches to the verification of hybrid systems. In: IEEE Conf. on Decision and Control, pp. 2867–2874 (2001)
35. Singh, A., Hespanha, J.: Models for gene regulatory networks using polynomial stochastic hybrid systems. In: CDC 2005 (2005)
36. Wehrle, M., Helmert, M.: The Causal Graph Revisited for Directed Model Checking. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 86–101. Springer, Heidelberg (2009)

An Axiomatic Memory Model for POWER Multiprocessors

Sela Mador-Haim¹, Luc Maranget², Susmit Sarkar³, Kayvan Memarian³,
Jade Alglave⁴, Scott Owens³, Rajeev Alur¹, Milo M.K. Martin¹,
Peter Sewell³, and Derek Williams⁵

¹ University of Pennsylvania

² INRIA Rocquencourt-Paris

³ University of Cambridge

⁴ University of Oxford

⁵ IBM Austin

Abstract. The growing complexity of hardware optimizations employed by multiprocessors leads to subtle distinctions among allowed and disallowed behaviors, posing challenges in specifying their memory models formally and accurately, and in understanding and analyzing the behavior of concurrent software. This complexity is particularly evident in the IBM® Power Architecture®, for which a faithful specification was published only in 2011 using an operational style. In this paper we present an equivalent axiomatic specification, which is more abstract and concise. Although not officially sanctioned by the vendor, our results indicate that this axiomatic specification provides a reasonable basis for reasoning about current IBM® POWER® multiprocessors. We establish the equivalence of the axiomatic and operational specifications using both manual proof and extensive testing. To demonstrate that the constraint-based style of axiomatic specification is more amenable to computer-aided verification, we develop a SAT-based tool for evaluating possible outcomes of multi-threaded test programs, and we show that this tool is significantly more efficient than a tool based on an operational specification.

1 Introduction

Modern multiprocessors employ aggressive hardware optimizations to provide high performance and reduce energy consumption, which leads to subtle distinctions between the allowed and disallowed observable behaviors of multithreaded software. Reliable development and verification of multithreaded software (including system libraries and optimizing compilers) and multicore hardware systems requires understanding these subtle distinctions, which in turn demands accurate and formal models.

The IBM® Power Architecture®, which has highly relaxed and complex memory behavior, has proved to be particularly challenging in this respect. For example, IBM® POWER® is non-store-atomic, allowing two writes to different locations to be observed in different orders by different threads; these order variations are constrained by coherence, various dependencies among instructions,

and several barrier instructions, which interact with each other in an intricate way. The ARM architecture memory ordering is broadly similar.

Several previous attempts to define POWER memory consistency models [CSB93, SF95, Gha95, AAS03, AFI⁺09, AMSST0] did not capture these subtleties correctly. A faithful specification for the current Power Architecture was published only in 2011 [SSA⁺11] using an operational style: a non-deterministic abstract machine with explicit out-of-order and speculative execution and an abstract coherence-by-fiat storage subsystem. This specification was validated both through extensive discussions with IBM staff and comprehensive testing of the hardware. The operational specification has recently been extended to support the POWER load-reserve/store-conditional instructions [SMO⁺12].

This paper presents an alternative memory model specification for POWER using an axiomatic style, which is significantly more abstract and concise than the previously published operational model and therefore better suited for some formal analysis tools. One of the main challenges in specifying an axiomatic model for POWER is identifying the right level of abstraction. Our goal was to define a specification that is detailed enough to express POWER’s complexity, capturing the distinctions between allowed and disallowed behaviors, yet abstract enough to be concise and to enable understanding and analysis.

Our approach splits instruction instances into multiple abstract events and defines a happens-before relation between these events using a set of constraints. The specification presented here handles memory loads and stores; address, data, and control dependencies; the `isync` instruction, and the lightweight and heavyweight memory barrier instructions `lwsync` and `sync`. The specification does not include load-reserve and store-conditional instructions, mixed-size accesses, or the `eieio` memory barrier.

We show that this specification is equivalent to the existing POWER operational specification (permitting the same set of allowed behaviors for any concurrent POWER program) in two ways. First, we perform extensive testing by checking that the models give the same allowed behaviors for a large suite of tests; this uses tools derived automatically from the definitive mathematical statements of the two specifications, expressed in Lem [OBZNS11]. We also check that the axiomatic specification is consistent with the experimentally observed behavior of current IBM® POWER6® / IBM® POWER7® hardware. We then provide a manual proof of the equivalence of the operational and axiomatic specifications, using executable mappings between abstract machine traces and axiomatic candidate executions, both defined in Lem. We have checked their correctness empirically on a small number of tests, which was useful in developing the proof.

Finally, we demonstrate that this abstract constraint-based specification is useful for computer-aided verification. The testing described above shows that it can be used to determine the outcomes of multi-threaded test programs more efficiently than the operational model. This efficiency enables a tool to calculate the allowed outcomes of 915 tests for which the current implementation of the operational tool [SSA⁺11] does not terminate in reasonable time and space.

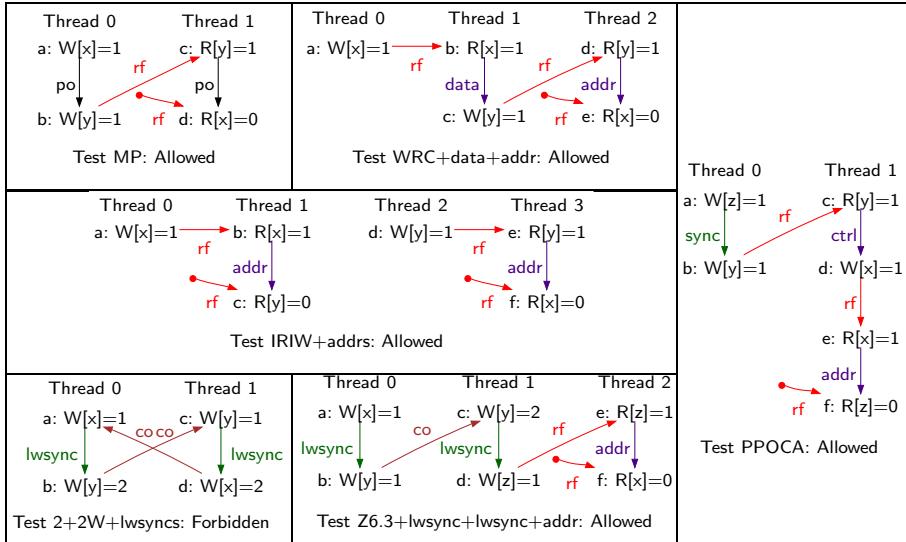


Fig. 1. Examples illustrating the POWER memory model

Further, the axiomatic specification lends itself to a SAT/SMT-solving approach: a hand-coded translation of the axiomatic model using minisat [ES05] reduces execution time for the full test suite radically, from 82 CPU-days to 3 hours.

The full definitions of our specifications, test suite, test results, and proof are available in on-line supplementary material [Sup].

2 Background: The POWER Memory Model

This section highlights some of the subtleties of the POWER memory model, and then overviews its abstract-machine semantics, referring to Sarkar et al. [SSA⁺11, SMO⁺12] for a complete description.

2.1 Subtleties of the POWER Memory Model

Figure 1 shows example candidate executions for several concurrent litmus tests, MP, WRC+data+addrs, etc. (each test is defined by its assembly source code and its initial and final register and memory state, which select a particular execution). For each, the diagram shows a graph with nodes for memory reads or writes, each with a label (a, b, \dots), location (x, y, \dots), and value ($0, 1, \dots$). The edges indicate program order (po), data dependencies from a read to a write whose value was calculated based on the value read (data), address dependencies from a read to a read or write whose address was calculated based on the value read (addr), control dependencies from a read to instructions following a conditional branch whose condition involved the value read (ctrl), and lwsync

and `sync` barriers. The *reads-from* edges (`rf`) go from a write (or a dot indicating the initial state) to any read that reads from this write or initial value. *Coherence* (`co`) edges give, for each location, a total order over the writes to that location. Register-only and branch instructions and are elided.

Relaxed memory behavior in POWER arises both from out-of-order and speculative execution within a hardware thread and from the ways in which writes and barriers can be propagated from their originating thread to other threads. For writes and reads to different addresses, absent any barriers or dependencies, the behavior is unconstrained. The message-passing MP example illustrates this behavior: the writes `a` and `b` might commit in either order, then propagate to Thread 1 in either order, and reads `c` and `d` can be satisfied in either order. Any of these effects can give rise to the given execution. To prevent them, a programmer could add an `lwsync` or `sync` barrier between the writes (making their commit order and their propagation order respect program order) and either add a barrier between the reads or make the second read address-dependent on the first (perhaps using the result of the first read xor'd with itself to form the address of the second read, introducing an artificial dependency) thus ensuring that it cannot be satisfied until the first read is satisfied. For example, test MP+`lwsync+addr` (a variation of MP with a `lwsync` edge in one thread and an `addr` edge in the other) is forbidden. A control dependency alone does not prevent reads being satisfied speculatively; the analogous MP+`lwsync+ctrl` is allowed. But adding an `isync` instruction after a control dependency does: MP+`lwsync+ctrlisync` is forbidden.

Dependencies have mostly local effects, as shown by the WRC+`data+addr` variant of MP, where the facts that `b` reads from `a`, and that `c` is dependent on `b`, are not sufficient to enforce ordering of `a` and `c` as far as Thread 2 is concerned. To enforce that ordering, one has to replace the data dependency by an `lwsync` or `sync` barrier. This example relies on the so-called *cumulative* property of the barriers, which orders all writes that have propagated to the thread of the barrier before all writes that occur later in program order, as far as any other thread is concerned.

The independent-reads-of-independent-writes IRIW+`addrs` example shows that writes to different addresses can be propagated to different threads (here Threads 1 and 3) in different orders; POWER is not *store-atomic*, and thread-local reorderings cannot explain all its behaviors. Inserting `sync` instructions between the load instructions on Thread 1 and Thread 3 will rule out this behavior. Merely adding `lwsyncs` does not suffice.

Returning to thread-local reordering, the PPOCA variant of MP shows a subtlety: writes are not performed speculatively as far as other threads are concerned, but here `d`, `e`, and `f` *can* be locally performed speculatively, before `c` has been satisfied, making this execution allowed.

The two final examples illustrate the interplay between the coherence order and barriers. In test Z6.3+`lwsync+lwsync+addr` (blw-w-006 in [SSA+11]), even though `c` is coherence-ordered after `b` (and so cannot be seen before `b` by any thread), the `lwsync` on Thread 1 does not force `a` to propagate to Thread 2 before `d` is (in the terminology of the architecture, the coherence edge does not

bring b into the *Group A* of the Thread 1 lwsync). This test outcome is therefore allowed. On the other hand, some combinations of barriers and coherence orders are forbidden. In $2+2W+\text{lwsyncs}$, for example, there is a cycle among the writes of coherence and lwsync edges; such an execution is forbidden.

2.2 The Operational Specification

The operational specification of Sarkar et al. [SSA⁺11, SMO⁺12] accounts for all these behaviors (and further subtleties that we do not describe here) with an abstract machine consisting of a set of threads composed with a storage subsystem, communicating by exchanging messages for write requests, read requests, read responses, barrier requests, and barrier acknowledgments (for sync). Threads are modeled with explicit out-of-order and speculative execution: the state of each thread consists of a tree of in-flight and committed instructions with information about the state of each instruction (read values, register values etc.) and a set of unacknowledged syncs. The thread model can perform various types of transitions. Roughly speaking (without detailing all the transition preconditions), a thread can:

- Fetch an instruction, including speculative fetches past a branch.
- Satisfy a read by reading values from the storage subsystem or by forwarding a value from an in-flight write. Reads can be performed speculatively, out-of-order, and (before they are committed) can be restarted if necessary.
- Perform an internal computation and write registers.
- Commit an instruction (sending write and barrier requests to the storage subsystem).

The state of the storage subsystem consists of a set of (1) writes that have been committed by a thread, (2) for each thread, a list of the writes and barriers propagated to that thread, (3) the current constraint on the coherence graph as a partial order between writes to the same location, with an identified linear prefix for each location of those that have *reached coherence point*, and (4) a set of unacknowledged syncs. The storage subsystem can:

- Accept a barrier or write request and update its state accordingly.
- Respond to a read request.
- Perform a partial coherence commit, non-deterministically adding to the coherence graph an edge between two as-yet-unrelated writes to the same location.
- Mark that a write reached a coherence point, an internal transition after which the coherence predecessors of the write are linearly ordered and fixed.
- Propagate a write or a barrier to a thread, if all the writes and barriers that are required to propagate before it have been propagated to this thread. A write can only be propagated if it is coherence-after all writes that were propagated to a thread, but the abstract machine does not require all writes that are coherence-before it have been propagated, thus it allows some writes, which might never be propagated to some of the threads, to be skipped.
- Send a sync acknowledgment to the issuing thread, when that sync has been propagated to all other threads.

3 The Axiomatic Specification

This section introduces our new specification of the POWER memory model in an axiomatic style. We begin by defining the semantics of a multithreaded program as a set of axiomatic candidate executions. We then give an overview of the axiomatic specification, which defines whether a given axiomatic candidate execution is consistent with the model, show how the examples in Section 2 are explained using this model, and finally provide the formal specification of the model.

3.1 Axiomatic Candidate Executions

We adopt a two-step semantics, as is usual in axiomatic memory models, that largely separates the instruction execution semantics from the memory model semantics by handling each individually.

We begin with a multithreaded POWER program, or litmus test, in which each thread consists of a sequence of *instructions*. Such a program may nondeterministically display many different behaviors. For example: reads may read from different writes, conditional branches may be taken or not taken, and different threads may write to the same address in a different order. During the execution of a program, any given static instruction may be iterated multiple times (for example, due to looping). We refer to such an instance of the dynamic execution of an instruction as an *instruction instance*.

To account for the differing ways a given litmus test can execute, we define the semantics of a multithreaded program as set of *axiomatic candidate executions*. Informally, an axiomatic candidate execution consists of (1) a set of *axiomatic instruction instances*, which are instruction instances annotated with additional information as described below, and (2) a set of relations among these axiomatic instruction instances (in what follows, we will refer to instruction instances or axiomatic instruction instances for load and store instructions as *reads* and *writes*, respectively). An axiomatic candidate execution represents a conceivable execution of the program and accounts for the effects of a choice of branch direction for each branch instruction in the program, a possible coherence order choice, and a reads-from mapping showing which write a given read reads from.

An *axiomatic instruction instance* is an instruction instance of the program annotated with a thread id and some additional information based on instruction type. Axiomatic instruction instances are defined only for reads, writes, memory barriers (`sync`, `lwsync`, `isync`), and branches. Reads are annotated with the concrete value read from memory, while writes are annotated with the value written to memory. Barriers and branches have no additional information. Other instructions may affect dependency relationships in the axiomatic candidate execution, but are otherwise ignored by the model.

An axiomatic candidate execution consists of a set of axiomatic instruction instances, and the following relations between those axiomatic instruction instances:

- A *program order* relation po , providing a total order between axiomatic instruction instances in each thread.
- A *reads-from* relation rf , relating writes to reads to the same address.
- A *coherence* relation co , providing, for each address, a strict total order between all writes to that address.
- A *data dependency* relation from reads to those writes whose value depends on the value read.
- An *address dependency* relation from reads to those reads or writes whose address depends on the value read.
- A *control dependency* relation from each read to all writes following a conditional branch that depends on the value read.

For each candidate execution of a given program, the following conditions must hold: (1) for each thread, the sequence of instruction instances ordered by po agrees with the local thread semantics of that program, when running alone with the same read values; (2) for each read and write related by rf , the read reads the value written by the write; and (3) if a read is not associated with any write in rf , it reads the initial value.

As an example, consider the test MP+lwsync+ctrl, a variation of MP with an lwsync and a control dependency. The POWER program for this test is listed below:

Thread 0	Thread 1
(a) li r1,1	(f) lwz r1,0(r2)
(b) stw r1,0(r2)	(g) cmpw r1,r1
(c) lwsync	(h) beq LC00
(d) li r3,1	LC00:
(e) stw r3,0(r4)	(i) lwz r3,0(r4)

Instruction instances are not defined for register-only instructions. Therefore, there are no instances of instructions a, d and g in this example. The conditional branch h in the program may be either taken or not taken, but in this case it jumps to i, so in both cases the axiomatic candidate execution contains a single instance for each of the instructions: {b,c,e,f,h,i}. The program order po in all candidate executions of this program is the transitive closure of the set {(b,c),(c,e),(f,h),(h,i)} and the control dependency is {(f,i)}. The coherence order co is empty in this example because each write writes to a different address. Each of the two reads in Thread 1 may either read from the matching write in Thread 0 or from the initial value. Hence, there are four possible rf relations, and four axiomatic candidate executions.

3.2 Overview of the Specification

The axiomatic specification defines whether a given axiomatic candidate execution is *consistent* or not. This section provides an overview of our axiomatic POWER memory specification (formally described in Section 3.4).

For each axiomatic candidate execution, we construct a set of *events* that are associated with the axiomatic instruction instances, together with several relations over those events. These events and relations (as described below) capture the subtleties of the POWER memory model, including speculative out-of-order execution and non-atomic stores. The events and relations determine whether an axiomatic candidate execution is *consistent*. In more detail:

Uniprocessor Correctness Condition. The relations rf and co must not violate uniprocessor execution order, in the following sense: a read is allowed to read a local write only if that write precedes the read in program order, that write is the most recent (w.r.t. program order) write to that address in that thread, and there is no program-order-intervening read that reads from a different write (from another thread). Based on rf and co , we define fr as the relation from any read to all writes which are coherence-after the write that the read reads from. We define the communication relation $comm$ as the union of rf , fr and co . The uniprocessor condition requires that the transitive closure of $comm$ does not contain any edge which goes against program order.

Local Reordering. The effects of out-of-order and speculative execution in POWER are observable, as shown by the MP variations in Fig. II (including PPOCA). Reads can be satisfied speculatively and speculative writes can be forwarded to local reads, although not to other threads. The specification captures this by defining *satisfy* read events, *initiate* write events, and *commit* events for both reads and writes: a read is satisfied when it binds its value, and committed when it cannot be restarted and that value is fixed; a write is initiated when its address and value can be (perhaps speculatively) calculated and it can be propagated to thread-local reads and committed when it can propagate to other threads.

Non-atomic Stores. Writes in POWER need not be propagated to all other processors in an atomic fashion, as illustrated by WRC+data+addr (the write to x propagates to Thread 1 before propagating to Thread 2). As in the operational model (and previous axiomatic models [Int02, YGLS03]) to capture this behavior, we split each write into multiple *propagation* events. In our model each thread other than its own has a propagation event, whereas in the operational model some write propagations can be superseded by coherence-later propagations. A write propagating to a thread makes it eligible to be read by that thread.

Barriers and Non-atomic Stores. The semantics of the sync and lwsync barriers in POWER are quite subtle. As seen in WRC+lwsync+addr, lwsync has a cumulative semantics, but adding lwsync between every two instructions does not restore sequential consistency, as shown by IRIW+lwsyncs. As in the operational model, we capture this behavior by splitting barriers into multiple propagation events, analogous to those for writes, with the proper ordering rules for these. A barrier can propagate to a thread when all the writes in the cumulative Group A of the barrier have propagated to that thread.

Barriers and Coherence. As shown by Z6.3+lwsync+lwsync+addr, coherence relationships between writes do not necessarily bring them into the cumulative

Group A of `lwsync` barriers (or for that matter of `sync` barriers). We capture this behavior by allowing writes that are not read by a certain thread to propagate to that thread later than coherence-after writes. This weakened semantics for coherence must be handled with caution, and additional constraints are required to handle certain combinations of barriers and coherence edges, as shown by example 2+2W+lwsyncs (in Fig. 11).

To summarize, the specification uses the notion of events with possibly multiple events corresponding to an axiomatic instruction instance to capture these behaviors. There are four types of events:

1. *Satisfy events.* There is a single read satisfy event $sat(x)$ for each read axiomatic instruction instance x , representing the point at which it takes its value. Unlike in the operational model (in which a read might be satisfied multiple times on speculative paths), there is exactly one satisfy event for each read axiomatic instruction instance.
2. *Initiate events.* Each write has an initiate event $ini(x)$, the point at which its address and value are computed, perhaps speculatively, and it becomes ready to be forwarded to local reads.
3. *Commit events.* Each axiomatic instruction instance x of any type has a commit event $com(x)$. Reads and writes can commit only after they are satisfied/initiated. Writes and barriers can propagate to other threads only after they are committed.
4. *Propagation events.* For each write or barrier instruction x and for each thread t which is not the originating thread of x , there is a propagation event $pp_t(x)$, which is the point at which x propagates to thread t .

The main part of our axiomatic model is defined using *evord*, a happens-before relation between events, which must be acyclic for consistent executions. Given an axiomatic candidate execution, *evord* is uniquely defined using the rules listed below.

Intra-instruction Order Edges. Our specification provides two ordering rules that relate events for the same instruction: *events-before-commit* states that reads must be satisfied and writes must be initiated before they commit; *propagate-after-commit* states that an instruction can propagate to other threads only after it is committed.

Local Order Edges. The *local-order* rules for *evord* relate *sat*, *ini* and *com* events within each thread. For a pair of events x and y from program-ordered axiomatic instruction instances (x before y), these events must occur in program order and cannot be reordered in the following cases:

- x is a read satisfy event and y is a read satisfy or write initiate event of an instruction with either an address or data dependency on x .
- x and y are read satisfy events separated by `lwsync` in program order.
- x and y are read or write commit events of instructions that have either data, address, or control dependency between them.
- x and y are read or write commit events for instructions accessing the same address.

- x and y are commit events and at least one of x and y is a barrier.
- x is a conditional branch commit event and y is a commit event.
- x and y are read or write commit events and there is a program-order-intervening instruction whose address depends on x .
- x is a read commit event, y is a read satisfy event, and both reads accessing the same address but reading from different non-local writes.
- x is a commit event of `sync` or `isync` and y is a read satisfy event.

Communication Order Edges. Communication rules order reads and writes to the same address from different threads, based on the relations rf , co , and fr . The *read-from* rule ensures that a read is satisfied only after the write it reads from propagates to the reading thread. The *coherence-order* rule states that for a write w , any write w' that is coherence-after w can propagate to w 's thread only after w commits. The *from-read* rule defines which writes can be observed by each read. It states that if a read r reads from w , any write w' that is coherence-after w can propagate only to the thread of r after r is satisfied.

One implication of the above definition for the from-read rule is that the reads in each thread can only observe writes in coherence order, even if they propagate out-of coherence order. For example, if w_1 is coherence-before w_2 , and w_1 propagates to t after w_2 , any program-order-later read in t would still read from w_2 and not w_1 .

Intra-thread Communication Edges. If a read receives a value written by a local write, this read must be satisfied after the write is initiated. This edge is the only type of intra-thread communication edge in this specification. There are no *evord* edges arising from *fr* or *co* edges between events for axiomatic instruction instances in the same thread.

Before Edges (Barrier Cumulativity). In the operational specification, any write reaching thread t before a barrier is committed in t must (unless superseded by a coherence successor) be propagated to any other thread before that barrier is propagated, as shown in the WRC test. Our axiomatic specification expresses the same cumulative property using a *before-edge* rule, stating that if a write propagates to a thread before a barrier commits or vice versa (a barrier propagates before a write commits), then the propagation events for these two instructions must have the same order between them in any other thread.

Before edges apply both to events associated with instructions from the same thread and instructions from different threads. For same-thread instructions, they require writes separated by a barrier to propagate to other threads in program order. For instructions from different threads, their effect is enforcement of a global propagation order between writes and barriers that are related by communication edges.

After Edges (Sync Total Order). Heavyweight `syncs` are totally ordered. A `sync` may commit only after all previously committed `syncs` in the program have finished propagating to all threads. We enforce this using the *after-edge* rule, which states that if a `sync` b propagated to a thread after committing a local `sync` a , then any event associated with b must be ordered after any event associated

with a . Note that this is different from (and simpler than) the operational model, where `sync` propagations can overlap.

Extended Coherence Order. The extended coherence order, $cord$, is a relation between axiomatic instruction instances that includes co , as well as edges from each write w to each barrier b that commit after w propagates to b 's thread, and from each barrier b to each write w that commits after b propagates to w 's thread. Extended coherence must be acyclic, which captures the coherence/lwsync properties of examples such as `2+2W+lwsyncs`.

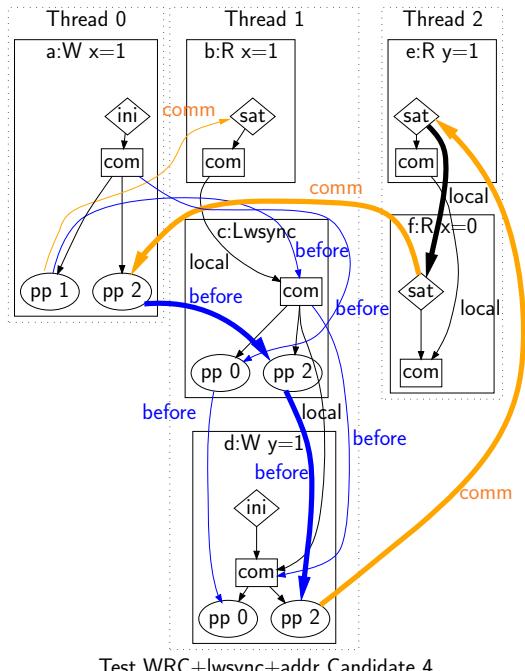
3.3 Examples

We now discuss how the examples in Fig. II are explained by this model, principally by looking at the *evord* relation between the events in each litmus test.

MP. In this example, there is a *read-from* edge from the pp_1 of b to the *sat* of c , and a *from-read* edge from the *sat* of d to the pp_1 of a . Without any additional dependencies or barriers, there are no additional edges and no cycles. Adding a dependency between the two reads on Thread 1 would add a local edge between the *sat* events of these reads. Adding an `lwsync` between the writes in Thread 0 would add local edges from a to the barrier to b and *before* edges between the pp_1 events of these three instruction instances, forming the cycle: $pp_1(a) \rightarrow pp_1(b) \rightarrow sat(c) \rightarrow sat(d) \rightarrow pp_1(a)$, making this forbidden.

WRC+lwsync+addr. The *evord* for this test is shown on the right. In this example, a is read by b , leading to a communication edge between them. As a result, the pp_1 event of a precedes the `lwsync` barrier, triggering a *before-edge* between them and forcing a to propagate before c in Thread 2. The result is a cycle: $pp_2(a) \rightarrow pp_2(c) \rightarrow pp_2(d) \rightarrow sat(e) \rightarrow sat(f) \rightarrow pp_2(a)$. Without the barrier, there would be no *before* edges connecting the propagation events of a and c . These two writes would be allowed to propagate to Thread 2 in any order, hence this example would be allowed.

IRIW+addrs. In this example, without barriers there would be no edges between the propagation events of a and d



and therefore they could be observed in any order by Threads 1 and 3. Adding `lwsync` between the reads in this example adds *before* edges from `a` to the barrier in Thread 1 and from `d` to the barrier in Thread 3, but there are no edges connecting the propagation events of the two writes yet. Replacing `lwsync` with a heavy-weight `sync`, however, would add *after* edges from the propagation event of the sync in Thread 1 to the sync in Thread 3, as well as as *after* edges from the sync in Thread 3 to Thread 1, and therefore there would be a cycle.

PPOCA. In this example, there are local edges between the *com* events of Thread 1. For the *sat* and *ini* events, there are edges between `d` and `e` (intra-thread communication) and between `e` and `f` (local), but control dependency does not add edges between the satisfying `c` to the initiate of `d`, and hence there is no cycle.

Z6.3+lwsync+lwsync+addr. In this test, the only communication edge connecting Thread 0 and Thread 1 is from the *com* of `b` to *pp₀* of `c` (due to coherence). This edge does not generate any *before* edges because it does not order any propagation event of Thread 0 before the barrier in Thread 1. Therefore, there are no edges between the propagation events of `a` and `d`, and hence there is no cycle.

2+2W+lwsyncs. In this example, there are *before* edges between `a` and `b` and between `c` and `d`, due to the barriers in these threads. Furthermore, the coherence order between `b` and `c` creates a communication edge between the *com* of `b` to the *pp₀* of `c` (and similarly for `d` and `a`). These edges do not form a cycle in *evord*. However, the before-edges and coherence relation form a cycle in *cord*, and therefore this test is forbidden.

3.4 Formal Specification

The formal specification of the model, automatically typeset from the Lem [OBZNS11] definition, is shown in Figure 2. The following functions define the edges of *evord*: *local_order* defines which pairs of instructions form the local edges; the *events_before_commit* and *propagate_after_commit* order the events of the same instruction; *communication* defines the communication edges; *read_from_initiated* is the intra-thread communication edges; *fbefore* defines which instructions are ordered by before-edges, and *before_evord_closure* defines the actual edges; similarly, *fafter* and *after_evord_closure* define the after edges. The *evord* relation itself is defined as the least fixed point of *evord_more*, starting from *evord_base*. Also listed are the uniprocessor correctness rule, *uniproc*, and the *cord* relation, *cord_of*.

4 Experimental Validation

We establish confidence in our axiomatic specification experimentally, by checking that it gives the same allowed and disallowed behaviors as the operational model [SSA+11], using a large suite of tests designed to expose a wide variety

```

let uniproc ace =
(* Uniprocessor correctness condition *)
let comm = communication_of ace in
let commr = transitive_closure_of comm in
((\forall (x, y) \in commr. \neg ((y, x) \in ace.ace_po))
let local_order ace events =
(* local order rules: do not reorder if true *)
{\(ex, ey)|\forall ex \in events, ey \in events |
(instruction_of ex, instruction_of ey) \in ace.ace_po \wedge (
(is.sat ex \wedge is.ini ey \wedge (instruction_of ex, instruction_of ey) \in ace.ace_datadep) \vee
(is.sat ex \wedge (is.ini ey \vee is.sat ey) \wedge (instruction_of ex, instruction_of ey) \in ace.ace_addrdep) \vee
(is.com ex \wedge is.com ey \wedge same_addr ex ey) \vee
(is.com ex \wedge is.com ey \wedge (instruction_of ex, instruction_of ey) \in ace.ace_datadep) \vee
(is.com ex \wedge is.com ey \wedge (instruction_of ex, instruction_of ey) \in ace.ace_addrdep) \vee
(is.com ex \wedge is.com ey \wedge (instruction_of ex, instruction_of ey) \in ace.ace_ctrldep) \vee
(is.com ex \wedge is.com ey \wedge is.fence ex) \vee
(is.com ex \wedge is.com ey \wedge is.fence ey) \vee
(is.branch ex \wedge is.com ey) \vee
(is.com ex \wedge is.com ey \wedge (\exists ez \in events. (instruction_of ex, instruction_of ez) \in ace.ace_po \wedge
(instruction_of ez, instruction_of ey) \in ace.ace_po \wedge (instruction_of ex, instruction_of ez) \in ace.ace_addrdep)) \vee
(is.com ex \wedge is.read ex \wedge is.read_ext ace ey \wedge
same_addr ex ey \wedge \neg (same.read_from ace ex ey)) \vee
(is.lwsync ex \wedge is.read_satisfy ey) \vee
(is.com ex \wedge is.read ex \wedge is.read_satisfy ey \wedge (\exists ez \in events. (instruction_of ex, instruction_of ez) \in ace.ace_po \wedge
(instruction_of ez, instruction_of ey) \in ace.ace_po \wedge is.lwsync ez)) \vee
(is.com ex \wedge (is.sync ex \vee is.isync ex) \wedge is.read_satisfy ey))\}
let events_before_commit ace events =
{\(ex, ey)|\forall ex \in events, ey \in events | ((is.ini ex \vee is.sat ex) \wedge is.com ey \wedge instruction_of ex = instruction_of ey)\}
let propagate_after_commit ace events =
{\(ex, ey)|\forall ex \in events, ey \in events | (is.com ex \wedge is.propagate ey \wedge instruction_of ex = instruction_of ey)\}
let communication ace events comm =
{\(ex, ey)|\forall ex \in events, ey \in events |
(instruction_of ex, instruction_of ey) \in comm \wedge (
(is.read_satisfy ex \wedge is.propagate ey \wedge propagation.thread_of ey = SOME (thread_of ex)) \vee
(is.propagate ex \wedge is.read_satisfy ey \wedge propagation.thread_of ex = SOME (thread_of ey)) \vee
(is.write_commit ex \wedge is.propagate ey \wedge propagation.thread_of ey = SOME (thread_of ex)))\}
let read_from_initialized ace events =
{\(ex, ey)|\forall ex \in events, ey \in events |
(is.ini ex \wedge is.sat ey \wedge thread_of ex = thread_of ey \wedge (instruction_of ex, instruction_of ey) \in ace.ace_rf)\}
let bbefore ace events ex ey =
(is.write ex \wedge is.fence ey) \vee (is.fence ex \wedge is.write ey)
let bbefore_evord_closure ace events evord_0 =
{\(ex, ey)|\forall ex \in events, ey \in events | (\exists tid \in ace.ace_threads. relevant_to_thread ex tid \wedge relevant_to_thread ey tid) \wedge
bbefore ace events ex ey \wedge
(\exists (ex_1, ey_1) \in evord_0.
relevant_to_thread ex_1 (thread_of ey) \wedge relevant_to_thread ey_1 (thread_of ey) \wedge
instruction_of ex_1 = instruction_of ex \wedge instruction_of ey_1 = instruction_of ey))\}
let fafter ace events ex ey =
(is.sync ex \wedge is.sync ey \wedge ex < ey)
let fafter_evord_closure ace events evord_0 =
{\(ex, ey)|\forall ex \in events, ey \in events |
(\exists tid \in ace.ace_threads. relevant_to_thread ex tid \wedge relevant_to_thread ey tid) \wedge
fafter ace events ex ey \wedge
(\exists (ex_1, ey_1) \in evord_0. relevant_to_thread ex_1 (thread_of ex) \wedge
instruction_of ex_1 = instruction_of ex \wedge instruction_of ey_1 = instruction_of ey))\}
let evord_base ace events comm =
local_order ace events \cup
read_from_initialized ace events \cup
events_before_commit ace events \cup
propagate_after_commit ace events \cup
communication ace events comm
let evord_more ace events evord_0 =
bbefore_evord_closure ace events evord_0 \cup
fafter_evord_closure ace events evord_0 \cup
{\(ex, ey)|\forall (ex, ey) \in evord_0, (ey', ez) \in evord_0 | ey = ey'\}
let cord_of ace events evord =
let bbefore_cord = bbefore_cord_of ace events evord in
bbefore_cord \cup ace.ace_co

```

Fig. 2. Formal specification of POWER in Lem

of subtle behaviors. We also check that the model is sound with respect to the observable behavior of current POWER hardware.

Implementations. For the operational specification, we use the `ppcmem` tool [SSA¹¹], which takes a test and finds all possible execution paths of the abstract machine. For the axiomatic specification, we adapt the `ppcmem` front end to (straightforwardly) enumerate the axiomatic candidate executions of a test, then filter those by checking whether they are allowed by the definition of the axiomatic model. The kernel for both tools is OCaml code automatically generated from the Lem definition of the model, reducing the possibility for error.

Test Suite. Our test suite comprises 4480 tests, including the tests used to validate the operational model against hardware [SSA¹¹]. It includes the *VAR3* systematic variations of various families of tests (<http://www.cl.cam.ac.uk/users/pes20/ppc-supplemental/test6.pdf>); new systematic variations of the basic MP, S and LB tests, enumerating sequences of intra-thread relations from one memory access to the next, including address dependencies, data dependencies, control dependencies and identity of addresses. It also includes the tests of the *PHAT* experiment, used to validate the model of [AMSS10]; and hand-written tests by ourselves and from the literature. Many were generated with our `diy` tool suite from concise descriptions of violations of sequential consistency [AMSS11]. The tests and detailed results are available in the on-line supplementary material.

Results: Comparing the Axiomatic and Operational Models. We ran all tests with the implementations of both models. The preexisting tool for evaluating the operational model gives a verdict for only 3565 of the tests; the remaining 915 tests fail to complete by timing out or reaching memory limits. In contrast, the implementation of our axiomatic model gives a verdict for all 4480 of the tests. For all those for which the operational implementation gives a verdict, the operational and axiomatic specifications agree exactly.

Results: Comparing the Model to Hardware Implementations. We also used the test suite to compare the behavior of the axiomatic specification and the behavior of POWER6 and POWER7 hardware implementations, as determined by extensive experimental data from the `litmus` tool (<http://diy.inria.fr/doc/litmus.html>). In all cases, all the hardware-observable behaviors are allowed by the axiomatic model. As expected, the model allows behaviors not observed in current hardware implementations, because our models, following the POWER architectural intent, are more relaxed in some ways than the behavior of any current implementation [SSA¹¹]. This result covers the 915 tests on which the operational model timed out, which gives evidence that the axiomatic model is not over-fitted to just the tests for which the operational result was known.

5 Proof of Equivalence to the Operational Specification

We establish further confidence in the equivalence of the axiomatic model presented here and the operational model, by providing a paper proof that the sets of behaviors allowed for any program are identical for both models: we show that any outcome allowed by the operational model is allowed in the axiomatic model and vice versa. In this section we provide an overview of the proof; the full proof is in the on-line supplementary material [Sup].

5.1 Operational to Axiomatic

The first part of the proof shows that any allowed test in the operational model is an allowed test in the axiomatic model. We do this by defining a mapping function $O2A$ from sequences of transitions of the operational model to a program execution and *evord* relations in the axiomatic model, proving that the resulting *evord* and *cord* are always acyclic.

A witness trace $W = \{tr_1, \dots, tr_n\}$ is a sequence of operational-model transitions, from an initial to a final system state. Given a witness W , our $O2A$ mapping generates a relation *evord'* by iterating over all labeled transitions. At each step, $O2A$ adds the corresponding events to *evord'*, and adds edges to the new event if they are allowed by *evord*. Most events correspond directly to certain transition types in the machine, with two exceptions: (1) *initiate-write* events do not correspond directly to any transition type, and are added to *evord'* either before the first forwarded read or before their commit; and (2) *irrelevant write propagation* events are write propagation events that do not correspond to any write propagation transition. These are added to *evord'* either before barriers (when required by before edges), or at end of the execution.

Another difference between the two models is in the handling of sync barriers. In the axiomatic model, *after* edges enforce a total order between syncs, effectively allowing syncs to propagate one at a time. In the operational model, a thread stops after a sync and waits for an acknowledgment that it propagated to all other threads, but several syncs can propagate simultaneously. When the mapping encounters a sync-acknowledge transition, it adds after-edges between this sync and all previously acknowledged syncs.

Theorem 1. *Given a witness $W = \{tr_1, \dots, tr_n\}$, the mapping $O2A(W)$ produces an axiomatic program execution with *co* and *rf* that satisfy the uniproc condition and acyclic *evord* and *cord* relations.*

We prove that *evord* for the axiomatic program execution is acyclic by showing that the *evord* relation produced by $O2A$ is both: (1) acyclic and (2) the same as the *evord* which is calculated from *co* and *rf*.

For all edges except the after edges, the *evord* produced by the mapping is acyclic by construction, because each newly added event is ordered after the previously added events. After edges are added between existing events when an acknowledge transition is encountered. The following Lemma guarantees that after edges do not form a cycle:

Lemma 1 (*Sync acknowledge for ordered sync propagations*). *If b_1 and b_2 are two sync instructions and b_1 is acknowledged before b_2 , then there is no path in $evord'$ from an event of b_2 to an event of b_1 .*

The mapping adds edges only if the corresponding $evord$ edges are allowed. To show that all the edges in $evord$ are in $O2A(W)$, we show that the mapping adds events in an order than agrees with the direction of the edges in $evord$. For each type of edge in $evord$, we show that the transition rules of the operational model guarantee this order.

5.2 Axiomatic to Operational

The second part of the proof shows that each allowed execution in the axiomatic specification is allowed by the operational specification. We define a mapping that takes the relations evaluated for the axiomatic specification, including rf , co , $evord$, and $cord$, and produces a sequence of transitions W for the operational specification.

Given an axiomatic candidate execution $CE = \{P, rf, co\}$ accepted by the axiomatic model, the $A2O$ mapping generates a witness $W = \{tr_1, \dots, tr_n\}$ for the operational model. The mapping takes $evord$ (which is acyclic for allowed executions), performs a topological sort of the events in $evord$, and then it processes these events in that order to produce W .

The $A2O$ mapping translates most events directly into corresponding transitions, with a few notable exceptions: (1) there are no transitions matching write initiate events; (2) write propagation events are allowed out-of-coherence-order, whereas write propagation transitions in the operational model must be in coherence order but some writes may be skipped, as identified by the mapping; (3) no event corresponds directly to sync acknowledge transitions (which are produced after a sync propagates to all threads); and (4) no events correspond to partial-coherence-commits, which are produced by the mapping according to co after processing write commit events.

Theorem 2. *Given an allowed candidate execution CE , the mapping $A2O(CE)$ produces a witness for an accepting path in the operational model.*

We prove this by induction on W . For each transition in W , we show that each type of transition is allowed based on the rules of $evord$ as well as $cord$ and the uniprocessor rule.

6 Evaluating the Axiomatic Specification with a SAT Solver

One advantage of the constraint-based axiomatic specification presented in this paper is that it can be readily used by constraint solvers (such as SAT or SMTs). To investigate this impact, we built a C++ implementation of the axiomatic specification using the minisat SAT solver [ES05]. Currently, this solver accepts

Table 1. Test suite runtime in the three checkers

model/tool	N	mean (s)	max (s)	effort (s)	memory
Operational/ppcmem	3565/4480	3016.19	2.4e+05	8.2e+07	40.0 Gb
Axiomatic/ppcmem	4480/4480	1394.14	2.3e+05	7.1e+06	4.0 Gb
Axiomatic/SAT	4188/4188	2.67	10.26	11170	—

diy sequential-consistency-violation cycles as input (rather than `litmus` sources), builds the corresponding tests internally and checks whether the resulting tests are allowed or forbidden. We ran this solver on the 4188 of the tests that were built from cycles.

We compare the execution time of this SAT-based tool to the `ppcmem` checkers for the operational and axiomatic specifications described in Section 4, which were built from Lem-derived code, emphasizing assurance (that they are expressing the models exactly as defined) over performance. They do not always terminate in reasonable time and space, so we resorted to running tests with increasing space limits, using 500+ cores in two clusters. In Table 1, N is the number of tests finally completed successfully in the allocated processor time and memory limits, w.r.t. the number tried. “mean” shows the arithmetic mean of the per-test execution time of successful runs; while “max” is the execution time for the test that took longest to complete successfully. The “effort” column shows the total CPU time allocated to running the simulators (including failed runs due to our resource limits). Finally, the “memory” column shows the maximum memory limit we used.

As shown by the last row of the table, the performance improvement of the SAT-based checker over either the operational or axiomatic versions of `ppcmem` is dramatic: the SAT solver terminates on all 4188 cycle-based tests, taking no more than about 10 seconds to run any test. The total computing effort is around 3 hours (wall-clock time is about 25 minutes on an 8-core machine) compared with the 82 CPU-days of the `ppcmem` axiomatic tool and 950 days of the `ppcmem` operational tool. One obtains similar results when restricting the comparison to the tests common to all three tools.

This efficient SAT-based encoding of the model opens up the possibility of checking properties of much more substantial example programs, e.g. implementations of lock-free concurrent data-structures, with respect to a realistic highly relaxed memory model.

Acknowledgments. The authors acknowledge the support of NSF grants CCF-0905464 and CCF-0644197; of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity; and funding from EPSRC grants EP/F036345, EP/H005633, and EP/H027351, ANR project ParSec (ANR-06-SETIN-010), ANR grant WMC (ANR-11-JS02-011), and INRIA associated team MM.

Legal. IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml. POWER, POWER6, POWER7, Power Architecture are registered trademarks of International Business Machines Corporation.

References

- [AAS03] Adir, A., Attiya, H., Shurek, G.: Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.* 14(5) (2003)
- [AFI⁺09] Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Zappa Nardelli, F.: The semantics of Power and ARM multiprocessor machine code. In: Workshop on Declarative Aspects of Multicore Programming (January 2009)
- [AMSS10] Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in Weak Memory Models. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 258–272. Springer, Heidelberg (2010)
- [AMSS11] Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: *Litmus*: Running Tests against Hardware. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 41–44. Springer, Heidelberg (2011)
- [CSB93] Corella, F., Stone, J.M., Barton, C.M.: A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM (1993)
- [ES05] Een, N., Sorensson, N.: Minisat - a SAT solver with conflict-clause minimization. In: International Conference on Theory and Applications of Satisfiability Testing (2005)
- [Gha95] Gharachorloo, K.: Memory consistency models for shared-memory multiprocessors. WRL Research Report 95(9) (1995)
- [Int02] Intel. A formal specification of Intel Itanium processor family memory ordering (2002), <http://developer.intel.com/design/itanium/downloads/251429.html>
- [OBZNS11] Owens, S., Böhm, P., Zappa Nardelli, F., Sewell, P.: Lem: A Lightweight Tool for Heavyweight Semantics. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 363–369. Springer, Heidelberg (2011)
- [SF95] Stone, J.M., Fitzgerald, R.P.: Storage in the PowerPC. *IEEE Micro* 15 (April 1995)
- [SMO⁺12] Sarkar, S., Memarian, K., Owens, S., Batty, M., Sewell, P., Maranget, L., Alglave, J., Williams, D.: Synchronising C/C++ and POWER. In: Programming Language Design and Implementation (2012)
- [SSA⁺11] Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Programming Language Design and Implementation (2011)
- [Sup] An axiomatic memory model for Power multiprocessors — supplementary material, <http://www.seas.upenn.edu/~selama/axiompower.html>
- [YGLS03] Yang, Y., Gopalakrishnan, G.C., Lindstrom, G., Slind, K.: Analyzing the Intel Itanium Memory Ordering Rules Using Logic Programming and SAT. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 81–95. Springer, Heidelberg (2003)

nuTAB-BackSpace: Rewriting to Normalize Non-determinism in Post-silicon Debug Traces*

Flavio M. De Paula¹, Alan J. Hu¹, and Amir Nahir²

¹ Dept. of Comp. Sci., Univ. of British Columbia, Canada

{depaulfm,ajh}@cs.ubc.ca

² IBM Corp. Haifa, Israel

nahir@il.ibm.com

Abstract. A primary challenge in post-silicon debug is the lack of observability of on-chip signals. In 2008, we introduced BackSpace, a new paradigm that uses repeated silicon runs to automatically compute debug traces that lead to an observed buggy state. The original BackSpace, however, required excessive on-chip overhead, so we next developed TAB-BackSpace, which uses only pre-existing on-chip debug hardware to compute an abstract debug trace with very low probability of error. With TAB-BackSpace, we demonstrated root-causing a (previously known) bug on an IBM POWER7 processor, in actual silicon.

The problem with these BackSpace approaches, however, is the need to repeatedly trigger the bug via the exact same execution. In practice, non-determinism makes such exact repetition extremely unlikely. Instead, what typically arises is an intuitively “equivalent” trace that triggers the same bug, but isn’t cycle-by-cycle identical. In this paper, we introduce nuTAB-BackSpace to exploit this observation. The user provides rewrite rules to specify which traces should be considered equivalent, and nuTAB-BackSpace uses these rules to make progress in trace computation even in the absence of exact trace matches. We prove that under reasonable assumptions about the rewrite rules, the abstract trace computed by nuTAB-BackSpace is concretizable — i.e., it corresponds to a possible, real chip execution (with the same low possibility of error as TAB-BackSpace). In simulation studies and in FPGA-emulation, nuTAB-BackSpace successfully computes error traces on substantial design examples, where TAB-BackSpace cannot.

1 Introduction

Post-silicon validation/debug is the problem of determining whether the fabricated chip of a new design is correct, and what is wrong if it behaves incorrectly. The problem lies between pre-silicon validation, which searches for design errors in models of the design before fabrication, and VLSI test, which searches for random manufacturing defects on each fabricated chip in high-volume production. Naturally, post-silicon validation/debug inherits characteristics from both,

* Supported in part by the Natural Sciences and Engineering Research Council of Canada.

but the differences necessitate novel solutions. Like pre-silicon validation, post-silicon validation focuses on *design errors*. The difference, however, is that the validation is of the actual silicon chip, which is roughly a billion times faster than simulation, can run the real software in the real system at full speed, and exhibits the true (not simulated) electrical and physical properties. Accordingly, post-silicon validation catches numerous bugs that escape pre-silicon validation, due to inadequate coverage, inaccurate models, approximate analyses, and mis-specified properties and constraints. Unfortunately, like VLSI test, post-silicon validation shares the problems of limited controllability and observability, as the internal signals on-chip are essentially inaccessible. Test and debug structures can be (and are) added on-chip, but any increase of the chip's area, power, or pins is expensive. These issues make post-silicon debugging extraordinarily challenging. Post-silicon debug currently consumes more than half of the total verification schedule on typical large designs, and the problem is growing worse [110].

Post-silicon validation/debug is broad and multi-faceted. To provide context, we briefly survey the overall debug flow and cite some representative research. Note that the post-silicon debug process is iterative, just like any other kind of debugging: at all stages of the process, the debug engineer formulates hypotheses about what might be going wrong, develops a test for the hypotheses, and then formulates new hypotheses based on the results. Because the focus is design errors, debug engineers typically have deep knowledge of the design.

The validation/debug process starts with test planning and stimulus generation: how to thoroughly exercise the die? This is analogous to simulation test-benches in pre-silicon validation, except that controllability/observability are limited to the pins and the test stimuli must be generated *quickly*. Typical tests include booting the OS, running applications, random instruction [1717], and focused test suites and exercisers for hard-to-verify parts of the design (e.g., [72]). To stress electrical bugs¹, these tests are run under a variety of system configurations and operating conditions (frequency, voltage, temperature, etc.).

When testing reveals the presence of a bug, the next step is to get a trace of what happened on the chip when the bug occurred. The challenge is the lack of observability, so the basic techniques are on-chip structures to improve observability, e.g., scan chains [29], trace buffers [274], and networks to access signals to record [2311]. Typically, one can take a snapshot of many/most latches of the design at a single cycle (scan), or record tens or hundreds of signals over a few hundred or thousand cycles (trace buffers), but getting this data off-chip is extremely slow and completely disrupts the test. Accordingly, the debug engineer has to trigger recording at exactly the right moment, and anecdotally, many debug engineers describe this as one of the most time-consuming tasks in

¹ For bugs that create functional errors, it's useful to distinguish between *logical bugs*, which could be replicated pre-silicon in the RTL, and *electrical bugs*, which result from electrical effects such as noise coupling, voltage droop, and timing errors, as some methods apply to only one or the other. This paper handles both. There are also electrical and physical bugs detected post-silicon that are not functional errors, e.g., power consumption, yield, reliability, etc., which we do not consider.

post-silicon debug. Most research supporting this phase of the debug process has focused on selecting signals that provide the best observability (e.g., [20,18,22,6]) harkening back to earlier work on observability for test (e.g., [19]). There is also research on *computing* debug traces: For example, assuming a deterministic test that is short enough to be simulated in its entirety on a deterministic fault-free model of the chip, it is possible to focus the trace buffer only on cycles where electrical errors are likely, relying on simulation to fill-in the fault-free cycles [3,30]. Closer to our work, IFRA [21] eliminates the assumptions of short tests and determinism, allowing a trace to be computed, for example, for a processor booting an operating system. The method works even if the error occurs very rarely, but is only for electrical bugs and is processor-specific. Our work builds on the BackSpace framework [14,15] (described below), which also computes traces of the full-speed silicon running long tests. The framework handles non-determinism and both logical and electrical bugs, but requires bugs to be reasonably repeatable.

Only when bug traces are available can debugging proceed. In a manual debug flow, the debug engineer finally has some insight into what is happening on-chip and can start ruling out possibilities and forming new hypotheses. Research results to support this process include automatically simplifying the bug trace [11,16], and using the trace to localize possible explanations [28,31], and even make layout repairs [10]. All of these methods depend on having traces showing what is happening on-chip leading up to the bug.

This paper focuses on the central task of deriving such debug traces, showing on-chip signals for many cycles leading up to an observed bug or crash. Until the trace is obtained, further debugging is essentially impossible.

1.1 The BackSpace Framework

Our work builds on the BackSpace framework, a novel paradigm that uses repeated silicon runs to automatically compute debug traces that lead to an observed buggy state. The core assumption of BackSpace is that the bring-up tests can be run repeatedly, and the bug being targeted will be at least somewhat repeatable (e.g., with probability $1/n$ for reasonably small n). The methods rely on repetition, which is fast on silicon, to compensate for the lack of observability.

The original BackSpace [14] introduced the basic theory and a proof-of-concept implementation on a small design. The method relied on some on-chip hardware, pre-image computations, and repetition to compute a provably correct trace to the bug. In theory, it solved the problem of computing a trace perfectly: computing arbitrarily long sequences of all signals on-chip, leading up to the bug. However, this perfect solution came with impractical overhead: correctness relied on computing breakpoints, signatures, and pre-images over the entire concrete state of the chip. The hardware overhead was too high to be practical.

Most complex chips, however, already include some on-chip debug hardware. In TAB-BackSpace [15], we flipped the problem around: instead of adding excessive on-chip hardware for a perfect debug solution, we leveraged the BackSpace approach to get much more out of the *already existing* in-silicon debug logic

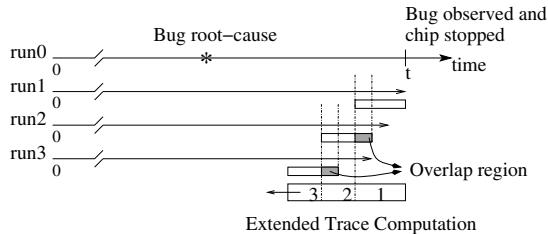


Fig. 1. TAB-BackSpacing. Once the bug is observed, we re-run the chip with trace arrays enabled, i.e., run1; we collect the information from the trace arrays and compute a new set of triggers for the subsequent run (run2); and we iterate these steps, extending the length of the computed trace beyond the trace arrays’ depth.

(i.e., trace buffers). Thus, there is no additional hardware cost. TAB-BackSpace achieves the effect of extending the trace buffer arbitrarily far back in time (assuming no spurious traces — more on this below).

Fig. 1 gives an overview of TAB-BackSpace. We assume the trace buffer records until stopped by a trigger. TAB-BackSpace iterates the following:

1. Run the chip until it “crashes” (hits the bug or the programmed breakpoint).
2. Dump out the state of the trace buffer into a file.
3. Select an entry from the trace dump as the new trigger condition, configuring the breakpoint circuitry to stop the chip when it hits this breakpoint on the next run.

The trace-buffer dump of the next run will overlap the most recent trace-dump by some number of cycles f . If all states in the overlapping region agree, we join the new trace-dump to the previous trace-dump, extending the length of the computed trace; if not, we select another state to be the breakpoint and try again. If the length of each trace-dump is m , then after n iterations, we will have computed a trace approximately $n(m - f)$ cycles long (approximate because f may vary between runs). Using TAB-BackSpace, we demonstrated root-causing a (previously known) bug on an IBM POWER7 processor, in actual silicon.

In theory, the weakness of TAB-BackSpace is the possibility of spurious abstract traces. By practical necessity, a trace buffer can record only a tiny fraction of on-chip signals. Therefore, the trace computed is an abstract trace. When two abstract trace dumps agree on the overlap region, TAB-BackSpace joins the two into a longer abstract trace, implicitly assuming that the underlying concrete traces agree as well, which might not be true. Empirically, we showed that by using a reasonably sized overlap region, the possibility of spurious traces could be made very small.

In practice, the real weakness of TAB-BackSpace is the need to repeatedly trigger the bug via the same execution. Non-determinism in the hardware and bring-up environment makes such exact repetition unlikely. The result is that the new trace-dump doesn’t completely agree with the previous trace over the entire overlap region, so TAB-BackSpace fails to make progress. Indeed, the

POWER7 result was achieved only by creating an environment that minimized non-determinism: running on bare metal, only one core enabled, and using a specialized post-silicon exerciser [15]. Creating an environment to minimize non-determinism while still triggering a bug is a difficult and time-consuming task.

We have observed, however, that although an *exact* match rarely occurs, what typically happens in practice is that the same bug is triggered by an intuitively “equivalent” trace, that isn’t cycle-by-cycle identical. How can we formalize the debug engineer’s informal notion of “equivalent”? And how do we extend TAB-BackSpace to correctly account for such user-specified equivalences?

This paper is an answer to those questions. The debug engineer provides rewrite rules to specify which traces should be considered equivalent, and our new algorithm uses those rules to make progress in trace computation even in the absence of exact trace matches. Under reasonable assumptions about the rewrite rules, and about the trace buffer length and signals, we prove that the abstract trace computed by nuTAB-BackSpace is concretizable — i.e., it corresponds to a possible, real chip execution. In simulation studies, we show that nuTAB-BackSpace can indeed compute correct error traces, even when non-determinism renders TAB-BackSpace infeasible. Finally, we demonstrate nuTAB-BackSpace successfully computing error traces on an industrial-size SoC in FPGA-emulation, where TAB-BackSpace cannot.

2 Background

2.1 Trace Buffers

A trace buffer is an on-chip structure for storing limited history of internal events that occur on-chip during full-speed execution. Because of the importance of post-silicon debug, most complex chips are now built with trace buffers.

A typical trace buffer consists of a memory array, organized as a FIFO, perhaps with some simple compression capabilities. A small number (typically tens to a few hundred) of important signals on the chip are routed to the FIFO. The signals routed to the trace buffer must be chosen before the chip is fabricated (although some limited reconfigurability is sometimes provided). The signals can be recorded in the FIFO in real-time as the chip runs, capturing typically a few hundred to a few thousand cycles of history. Control logic allows triggering the starting and stopping of this recording based on the signals that appear, cycle counters, watchdog timers, etc. There must also be some mechanism to read out (“dump”) the contents of the trace buffer, for example, by putting the chip into debug mode. Dumping the trace buffer is slow and radically perturb the execution of the chip, so debug methodologies avoid trying to continue an execution after a trace buffer dump.

In this paper, we assume very minimal trace buffer capabilities. We assume the recording can run continuously (the array treated as a circular buffer), and that we can set a breakpoint to stop recording when a specified input signal reaches the trace buffer. The trace buffer can be dumped arbitrarily later. This

gives the effect recording the last m cycles of the trace buffer signals before the chip “stops” at the breakpoint, where m is the length of the trace buffer.

2.2 Abstraction

We will reason about both the signals recorded in the trace buffer as well as the underlying state of the full chip as it runs in actual silicon. Because the signals recorded in the trace buffer are a subset of the total signals on-chip, we can view a state in the trace buffer as an abstraction of the state of the chip.

Formally, we model the full chip on-silicon as a finite-state transition system with state space S_c and (possibly non-deterministic) transition relation $\delta_c \in S_c \times S_c$. This is the *concrete* system. As is typical in model checking [12], we abstract away the inputs and consider only signals on-chip as the state. A *concrete trace* is a finite sequence of concrete states s_1, \dots, s_n such that $\forall i. (s_i, s_{i+1}) \in \delta_c$.

The choice of signals to record in the trace buffer defines an abstraction function $\alpha : S_c \rightarrow S_a$ that projects away everything but the chosen signals. S_a is the abstract state space, and the abstract transition relation $\delta_a(s_a, t_a)$ is defined as usual (e.g., [13]): $\exists s_c, t_c [\delta_c(s_c, t_c) \wedge s_a = \alpha(s_c) \wedge t_a = \alpha(t_c)]$. An *abstract trace* is a finite sequence of abstract states s_1, \dots, s_n such that $\forall i. (s_i, s_{i+1}) \in \delta_a$.

We lift the abstraction function to traces by abstracting each state of the trace: given a concrete trace σ_c , we get a unique abstract trace $\alpha(\sigma_c)$. In the opposite direction, an abstract trace σ_a is said to be *concretizable* if there exists a concrete trace σ_c such that $\sigma_a = \alpha(\sigma_c)$. Because the abstract transition relation is conservative, not all abstract traces are concretizable; such traces are called *spurious*. In practice, concretizability is a crucial property: a spurious trace doesn’t correspond to any possible execution of the real hardware, so it is not only wrong, but it misleads the debug engineer and wastes time.

2.3 Semi-Thue Systems

We will allow the debug engineer to specify intuitive notions of “equivalence” by providing rewrite rules. This provides ease-of-use, expressiveness, and a rich underlying theory that allows efficient checking of equivalent traces. In particular, we treat the debug trace and trace buffer dumps as strings whose alphabet is the abstract state space, and the user-provided rewrite rules produces a string rewriting system AKA a semi-Thue system. Semi-Thue systems have been extensively studied; our presentation is based on [95].

Definition 1. *A semi-Thue system is a tuple (Σ^*, \mathcal{R}) , where*

- Σ is a finite alphabet,
- \mathcal{R} is a relation on strings from Σ^* , i.e., $R \subseteq \Sigma^* \times \Sigma^*$.

Each element $(l, r) \in R$ is called a rewrite rule, notated as $l \rightarrow r$. Rewrite rules can be applied to arbitrary strings as follows: for any $u, v \in \Sigma^*$, $u \rightarrow v$ iff there exists an $(l, r) \in R$ such that for some $x, y \in \Sigma^*$, $u = xly$ and $v = xry$. The

notation \rightarrow^* is the reflexive and transitive closure of \rightarrow . We denote the symmetric closure of \rightarrow^* by \leftrightarrow^* , which is an equivalence relation on Σ^* .

The question we need to solve is whether two strings x and y are equivalent, i.e., whether $x \leftrightarrow^* y$. This is the standard “word problem” for semi-Thue systems. In general, the problem is undecidable, but under certain restrictions on the rewrite rules, the problem can be solved efficiently by reducing each of x and y to a unique normal form representing the equivalence class.

Definition 2. A semi-Thue system is Noetherian (terminating) if there is no infinite chain x_0, x_1, \dots such that for all $i \geq 0$, $x_i \rightarrow x_{i+1}$.

Noetherianness can be established by finding an ordering function (e.g., string length) that all rewrite rules obey. Under the assumption of Noetherianness, the two properties in the next definition are equivalent:

Definition 3. A semi-Thue system is **confluent** if for all $w, x, y \in \Sigma^*$, the existence of reductions $w \rightarrow^* x$ and $w \rightarrow^* y$ implies there exists a $z \in \Sigma^*$ such that $x \rightarrow^* z$ and $y \rightarrow^* z$. A semi-Thue system is **locally confluent** if for all $w, x, y \in \Sigma^*$, the existence of reductions $w \rightarrow x$ and $w \rightarrow y$ implies there exists a $z \in \Sigma^*$ such that $x \rightarrow^* z$ and $y \rightarrow^* z$.

A key result from rewriting theory is that for a rewriting system that is confluent and Noetherian, any object can be reduced to a unique normal form by applying rewrite rules arbitrarily until the object is irreducible (i.e., no rules apply). Furthermore, two objects are equivalent $x \leftrightarrow^* y$ iff their unique normal forms are the same. We will use the notation $N(x)$ to denote the unique normal form for any string x .

3 nuTAB-BackSpace

3.1 Formalizing the Intuition

Before describing the nuTAB-BackSpace algorithm, we first need to formalize our assumptions about the user-supplied abstraction and rewrite rules.

The fundamental principle underlying the BackSpace approaches is to use repetition to compensate for the lack of on-chip observability. The fundamental challenge, therefore, is how to determine when a new run of the chip is following “the same” execution as a previous one, so that information from the two physical runs can be combined.

The first technique is the breakpoint mechanism. We never try to combine traces unless the new trace breakpoints (i.e., the hardware reaches a specified state) on a state from the older traces. Because the two traces share an identical state, we are guaranteed that we can combine the two traces at that state and have a valid, longer trace — but the guarantee is only valid at the level of abstraction of the breakpoint state. In the original BackSpace, the breakpoint was concrete, guaranteeing that the algorithm constructed a valid, concrete trace leading to the bug. In TAB-BackSpace and nuTAB-BackSpace, the breakpoint

is only on a partial state, so the guarantee is only that the constructed trace is a legal, but possibly spurious (non-concretizable), abstract trace.

To reduce the possibility of spurious traces, and since a trace buffer provides multiple cycles of history anyway, we therefore insist that not only the breakpoint match, but every abstract state match in a multicycle overlap region between a new trace buffer dump and the previously computed trace. Intuitively, the longer the overlap region we require to match, the less likely that we compute spurious traces. We can formalize the intuition that a large enough overlap eliminates spurious traces as follows:

Definition 4. Let l_{div} (“divergence length”) be the smallest constant such that for all concrete traces $x_1y_1z_1$ and $x_2y_2z_2$ (where the xs , ys , and zs are strings of concrete states), if $\alpha(y_1) = \alpha(y_2)$ and the length $|\alpha(y_1)| > l_{div}$, then $x_1y_1z_2$ and $x_1y_2z_2$ are also valid concrete traces.

In other words, if two concrete executions share a long enough period of abstracting to the same states, then the future concrete execution is oblivious to what happened before that period, and so the combined abstract trace is not spurious. Note that the divergence length is specific to the design and also to the chosen abstraction function.

Although l_{div} may not always exist (because, for example, the abstraction function might abstract away key information from the concrete traces), in theory, it is straightforward to check whether the length of the overlapping region is longer than l_{div} : let f be the length of the overlapping region. Do there exist two traces $\sigma_1=x_1y_1z_1$ and $\sigma_2=x_2y_2z_2$ such that $|x_i|=|z_i|=1$, $|y_1|=|y_2|=f$, $\alpha(y_1)=\alpha(y_2)$, and either $x_1y_1z_2$ or $x_1y_2z_2$ are not valid traces? If not, we know that $f > l_{div}$. Otherwise, $f \leq l_{div}$. Therefore, all we need is to unroll the design (as in bounded model checking [8]) up to $f + 2$ cycles and check for a witness.

In practice, it may be unrealistic to unroll the design for $f + 2$ cycles. However, in [15] and in Section 4.1, we show that we can empirically limit the number of spurious traces. In particular, if we have trace dumps from different concrete executions that match on the overlap region, we dub this a “false match”, which is a necessary (but not sufficient) condition for a spurious trace. Our experiments show that false matches are rare when the overlap region is reasonably long.

Indeed, as noted earlier, the problem in practice is not too many matches generating spurious traces, but the lack of exact matches preventing any progress in trace computation. Empirically, however, we have often observed intuitively “equivalent” traces that are not cycle-by-cycle matches, e.g., a trace with slightly different timing, with independent events reordered, etc. These are all differences that could be manipulated via rewriting, so we propose to allow the debug engineer to specify rewrite rules to define what “equivalent” means to them, on a particular design. nuTAB-Backspace will then match overlap regions if they are equivalent under the specified rewriting, rather than requiring an exact match.

Will this idea produce correct traces? Correctness depends on the rewrite rules respecting the semantics of the design. Accordingly, we impose a few restrictions on the rewrite rules. Not surprisingly, we require that the rules be Noetherian and confluent, which allows efficient equivalence checking via reduction to the unique

normal form. To capture the notion that the rewrite rules truly reflect equivalent traces of the underlying concrete chip, we define the concept of concretization preservation:

Definition 5. Consider a rewrite rule $l \rightarrow r$ on strings of abstract states. The rewrite rule is **concretization preserving** if for all concrete states x_c and z_c , the concretizability of the abstract state sequence $\alpha(x_c)l\alpha(z_c)$ to a concrete sequence starting with x_c and ending with z_c implies the concretizability of the abstract state sequence $\alpha(x_c)r\alpha(z_c)$ to a concrete sequence starting with x_c and ending with z_c , i.e.:

$$\forall \text{concrete states } x_c, z_c \left[\begin{array}{c} (\exists \text{concrete trace } x_c y_l z_c . \alpha(y_l) = l) \\ \Rightarrow \\ (\exists \text{concrete trace } x_c y_r z_c . \alpha(y_r) = r) \end{array} \right]$$

Obviously, a rewrite rule should be rejected if it breaks concretizability altogether. This definition is slightly stronger in that it requires that a pre-existing concretization be preserved, *mutatis mutandis* the rewriting.

As with l_{div} , in theory, it is straightforward to check whether a rule is concretization preserving. There are a finite number of rewrite rules, $l \rightarrow r$, each of which is finite in length. Does there exist a concrete trace $x_c y_l z_c$ such that $\alpha(y_l) = l$, but where no string y_r exists such that $x_c y_r z_c$ is a concrete trace and $\alpha(y_r) = r$? One could, for example, use bounded model checking to enumerate all x_c and z_c that satisfy the antecedent of the definition, and then use bounded model checking to check that each satisfying x_c and z_c also satisfies the consequent.

In practice, depending on the design and abstraction, this check may also not be realistic. On the other hand, debug engineers have expert design knowledge, so they are capable of defining rewrite rules that are concretization preserving (or close enough for their purposes).

3.2 Algorithm

Algorithm 1 presents the nuTAB-BackSpace procedure: starting from a given crash state and its corresponding trace-buffer, it iteratively computes an arbitrarily long sequence of predecessor abstract states by going backwards in time. This procedure has 4 user-specified parameters: *steps_bound* specifies how many iterations back the algorithm should go; *retries_timeout* limits the amount of search for a new trace dump where the overlapping region with the trace computed so far is equivalent; the *time_bound* is a timeout for each chip-run and is a mechanism to tell whether a chip-run went on a path that does not reproduce the crash-state or buggy-state; and, *lbindex* is the trace buffer's smallest index, which defines a region either for the overlapping (TAB-BackSpace) or the normalization (nuTAB-BackSpace) of two consecutive trace buffers.

This procedure has 2 nested loops. The outer loop, lines 15 – 45, controls the three termination conditions for the algorithm: we reach the user-specified number of iterations; we reach the initial states; or the previous iteration was

unsuccessful. The outer loop is also responsible for joining the new trace buffer dump onto the successful trace computed so far (line 36), and then selecting a new state as the breakpoint for the next iteration. The inner loop, lines (20–35), is responsible for controlling the hardware while trying out different candidate-states, s_{cand} , given a *retries_timeout*. The procedure keeps track of time using the subroutine *ElapsedTime()* (passing *reset* as parameter resets the time counter, otherwise it counts the elapsed time since it was last reset). In each loop iteration, the procedure loads s_{cand} into the breakpoint-circuit (line 22), and runs the chip. The objective is to collect a new trace-buffer upon matching s_{cand} and match (after rewriting) it with the previous trace-buffer. If *ResetAndRun()* returns *TRUE* then the breakpoint circuitry matched s_{cand} and we have a new trace-buffer. Otherwise, the chip-run violates the *time_bound* parameter (line 24) because the current run took another path (caused by non-determinism). If the breakpoint occurs, we dump the contents of the trace-buffer for comparison with the trace computed so far. The *NormalizeAndCheck()* subroutine (line 28) computes the unique normal form of the overlapping region of the previously computed trace as well as the new trace dump, as described in Sec. 2.3, and then compares them to check equivalence. If the procedure neither breakpoints nor proves equivalence, *PickState()* (line 33) selects another candidate-state from the previous trace using a round-robin scheme while respecting *lindex* and the inner loop iterates. The procedure exits the inner loop when either it successfully proves equivalence of the overlapping regions of the two trace-buffers, or this loop has iterated longer than the specified *retries_timeout*.

3.3 Correctness

The main correctness theorem proves that the trace computed by Algorithm I is as informative as one could hope: it concretizes to a trace that leads to the actual crash state, using reachable states.

Theorem 1 (Correctness of Trace Computation). *If the rewriting rules are Noetherian, confluent, and concretization preserving, and if the size of all unique normal forms used to prove equivalence of overlapping regions is greater than l_{div} , then the trace produced by Algorithm I is concretizable to the suffix of a concrete trace leading from the initial states Q_0 to the crash state s .*

Proof: The proof is by induction on the iteration count i at the bottom of the outer loop. The base case is trivial, as when $i = 0$, the trace is a single trace buffer dump that ends at the crash state. Since this trace dump is taken from the physical chip, it can be concretized to the specific physical execution that occurred on-chip.

In the inductive case, let uy represent the trace computed so far, and let xv represent the new trace dump t_i , with $N(v) = N(u)$. In other words, u and v are the overlap region that has been proven equivalent by rewriting. By construction, x and y are non-empty.

We know that xv is concretizable to a trace with all states reachable from the initial states, because it is taken directly from the hardware. Therefore,

Algorithm 1. Crash State History Computation

```

1: input  $Q_0$  : set of initial states,
2:            $(s, t)$  : crash-state and trace-buffer
3:            $steps\_bound \in \mathbb{N}^+$  : user-specified bound on the number of iterations,
4:            $retries\_timeout \in \mathbb{N}^+$  : user-specified time-bound on retrials,
5:            $timebound$  : user-specified time bound for any chip-run
6:            $lbindex$ : user-specified lower-bound length of normal region;
7: output  $trace$  : equivalent sequence of abstract states;
8:  $i := 0$ ;
9: // initialize breakpointable candidate-state and current trace-buffer
10:  $i := 0$ ;  $s_{cand} := s$ ;  $t_i := t$ ;
11:  $trace := (t_i)$ ; // i.e., initialize trace with current trace-buffer
12: // initialize variable nindex; nindex gets updated by PickState()
13: // nindex range is  $[lbindex, |trace-buffer|]$ 
14:  $nindex := lbindex$ ;  $succ\_iteration := TRUE$ 
15: while ( $i < steps\_bound$ ) AND ( $s_{cand} \notin Q_0$ ) AND ( $succ\_iteration = TRUE$ ) do
16:    $equivalent := FALSE$ ;
17:    $matched := FALSE$ ;
18:   //Resets retrial elapsed time
19:    $ElapsedTime(reset)$ 
20:   while (! $equivalent$ ) AND ( $ElapsedTime(go) \leq retries\_timeout$ ) do
21:     // Program the hardware-breakpoint circuitry with  $s_{cand}$ 
22:      $LoadHardwareBreakpoint(s_{cand})$ ;
23:     // (Re-)run  $M'$  at full-speed with timeout  $timebound$ 
24:      $matched := ResetAndRun(timebound)$ ;
25:     if  $matched$  then
26:       // Dump trace-buffer contents  $t_i$ 
27:        $t_i := ScanOut()$ ;
28:        $equivalent := NormalizeAndCheck(t_i, t_{i-1}, nindex)$ ;
29:     end if
30:     if (! $matched$ ) OR (! $equivalent$ ) then
31:       // Pick another state following a round-robin scheme
32:       // and updates nindex
33:        $s_{cand} := PickState(nindex, t_{i-1})$ ;
34:     end if
35:   end while
36:   if  $equivalent = TRUE$  then
37:     // Accumulate trace
38:      $OverlapConcatenate(t_i, trace)$ ;
39:     // Pick a candidate-state in  $t_i$  for the next iteration
40:      $s_{cand} := PickState(nindex, t_i)$ ;
41:      $i := i + 1$ ;
42:   else
43:      $succ\_iteration := FALSE$ 
44:   end if
45: end while
46: return  $trace$ ;

```

$xN(v)$ has the same properties, by preservation of concretization. Similarly, uy is concretizable to a trace that leads to the crash state s , by the inductive hypothesis, and therefore, $N(u)y$ is, too, by preservation of concretization. Let $x_c v_c$ be a witness to the concretizability (with additional properties) of $xN(v)$, with $x = \alpha(x_c)$ and $N(v) = \alpha(v_c)$. Similarly, let $u_c y_c$ be a witness to the concretizability of $N(u)y$, with $N(u) = \alpha(u_c)$ and $y = \alpha(y_c)$.

From the hypotheses, $|N(u)| = |N(v)| > l_{div}$, so by the definition of l_{div} , both $x_c u_c y_c$ and $x_c v_c y_c$ are legal concrete traces. By construction, both start at reachable states, and therefore contain all reachable states. And both end at the crash state s . Therefore, either is a witness that the new trace computed by Algorithm II, $xN(u)y$, is concretizable to the suffix of a concrete trace leading from the initial states to the crash state. ■

4 Experiments

We present two experiments demonstrating the feasibility of nuTAB-BackSpace. In both, we compare our new method against TAB-BackSpace. We start with a simulation-based evaluation, where we have more controllability and can identify false matches. Then, we evaluate nuTAB-BackSpace on a hardware prototype.

4.1 Simulation-Based Evaluation

We use a router design (henceforth, the “Router”), which is an RTL implementation of a 4x4 routing switch. The Router is typically used by IBM for training new employees with IBM’s tools. The Router is a non-trivial design, but also not too complex to be simulated in its entirety. The design has 9958 latches, which is larger than most open-source design examples (e.g., from [25]).

The Router implements a routing policy, which is programmed beforehand in configuration registers. The Router routes incoming packets from four distinct input ports into one of four output ports. The Router recognizes packets in a pre-defined format containing source and destination addresses, payload, and bit-parity. In addition to routing the packets, the Router also checks the validity of incoming packets and rejects bad packets.

To simulate the Router, we use a constrained-simulation environment developed by IBM, using Cadence’s Incisive Simulator (with Specman Elite) v.09.20-2016. This proved very helpful when modeling environmental non-determinism.

We claim that when non-determinism cannot be extensively removed from the environment/design, TAB-BackSpace will either fail to produce a trace or will require an excessive number of re-trials. To validate this claim, we (1) set the TAB length to be 50 with no compression; (2) set the TAB width to 75 bits; (3) abstract three of the Router’s design blocks onto these 75 bits by using our architectural insight; (4) set a goal of 20 iterations for each “crash” state (setting $steps_bound = 20$ in Algorithm II); (5) for each iteration, we set a timeout of 5 hours to allow for a large number of re-trials when necessary (typically, each simulation-run takes about 10min); (6) and, randomly choose 30 abstract-states as our “crash” states.

To make sure the traces produced from these “crash” states are independent, we first generate a lengthy “crash” trace via constrained-random simulation. Then, we randomly choose 30 abstract-states, a_i , from this trace, with the following properties (Q_0 is the initial set of states): $\forall i, j. a_i, a_j \notin Q_0 \wedge (a_i \neq a_j) \wedge |i-j| > 1000$. These properties guarantee that the “crash” states in this experiment are far enough apart so that the computed traces are distinct. In other words, since $steps.bound = 20$ and each trace dump has 50 cycles, even if the overlap between two consecutive trace-dumps were one single cycle, the total number of cycles for each trace would be $20*49+1 < 1000$, which is smaller than the distance between two crash states.

In [15], we have empirically shown that, for the Router, an overlap of 30 cycles or more would most likely prevent false matches. Thus, in these experiments, we use 30 cycles as the lower bound for the overlapping region of two consecutive trace-buffers.

We use the constrained-simulation environment to simulate non-determinism. This environment provides many parameters to make each simulation run very different from one another. However, we want to control the non-determinism so that we have a fair comparison between TAB-BackSpace and nuTAB-BackSpace. Thus, we simulate non-determinism only affecting the delays on packet arrivals (a real scenario encountered in bring-up labs). We accomplish this behavior by changing the simulation environment such that it always uses a fixed random seed for everything except packet generation. For packet generation, we use an external and independent random generator to add different delays between packets in each run.

We need to provide a set of rules for normalizing the non-determinism during nuTAB-BackSpace simulations. In practice, defining rewrite rules will follow the same iterative process as debugging. In this case, we had worked with this design in [15] and had a good understanding of it. We had been unable to TAB-BackSpace the Router and this was due to non-determinism in the inter-packet delays, and so, in these experiments, we develop a set of rules to normalize such effects of non-determinism.

Recall that our abstract-model is based on 3 design blocks. In particular, three sets of signals in this abstract-model represent the same state-machine that is replicated across the 3 design blocks. In Fig. 2 we present one such machine. Notice that 6 states have self-loops, namely *idle*, *wait_buff*, *wait_data*, *wait_route*, *wait_idle*, *get_rest*. In [15], we observed that non-determinism in the inter-packet delays affects all these states with self-loop edges (e.g., a long delay might cause an input port to remain in *idle* or *wait_data* states for some number of cycles). The exception is the state *get_rest*. In this state, the Router processes incoming packets without interruption, that is, the Router does not accept partial packets. Thus, to normalize non-determinism, we define a rewriting system, $Router_{RS}(\Sigma^*, R)$.

Let $\text{Proj}(\cdot)_{sm}$ be a projection function that takes in an abstract-state, a , and projects it onto the set of bits representing the state machine from Fig. 2 and let $P = \{\text{idle}, \text{wait_buff}, \text{wait_data}, \text{wait_route}, \text{wait_idle}\}$. Now, we can define R as follows:

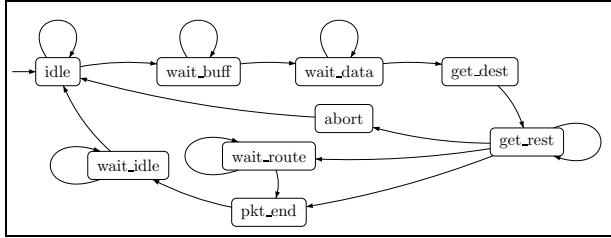


Fig. 2. Router’s Internal Packet-Processing State-Machine

$$\forall a. Proj(a)_{sm} \in P . aa \rightarrow a \quad (1)$$

Thus, this rewriting rule creates an equivalence class of traces, treating traces with different numbers of repetitions of certain states as similar.

Before we can apply nuTAB-BackSpace, we need to show that the rewriting system, $Router_{RS}(\Sigma^*, R)$, is Noetherian, confluent, and concretization-preserving. First, note that $Router_{RS}(\Sigma^*, R)$ is a length-reducing rewriting system, and so it is Noetherian. Next, note that Eq. 11 contains 5 rules (or technically, rule schema), and no two rules have overlapping left-hand sides. The only possible critical pairs arise from rewriting a string of the form aaa into aa with two different applications of a single rewrite rule. Obviously, these are locally confluent. Thus, the entire rewrite system, $Router_{RS}(\Sigma^*, R)$, is confluent. For concretization preservation, we have only an informal argument. Based on our knowledge of the design, any execution of the system that goes through a state that projects to P can spend more or less time in that state, without impacting the rest of the execution. This is exactly the property that concretization preservation captures. In contrast, when the state machine is in the state get_rest , the underlying concrete state tracks the number of cycles for the packet, so a rule that changed the number of get_rest cycles would not be concretization-preserving.

We deem a TAB-BackSpace iteration successful when two consecutive trace-buffers agree cycle-by-cycle over all 30 cycles, i.e., a full-overlap match; and a nuTAB-BackSpace is successful when the normalization-region (30 cycles or more) from the consecutive trace-buffers are equivalent under $Router_{RS}(\Sigma^*, R)$.

The experiments are successful. In Table 11, we show that nuTAB-BackSpace computes, for all crash-states, longer traces than TAB-BackSpace. Moreover, TAB-BackSpace could not compute even one iteration for 1/3 of the cases. And, when TAB-BackSpace is comparable to nuTAB-BackSpace with respect to the number of successful iterations (e.g., crash states 19, 27-30), nuTAB-BackSpace requires, for the most cases, an order of magnitude smaller number of runs.

4.2 Case Study: The Leon3 SoC Hardware Prototype

To demonstrate that nuTAB-BackSpace is feasible in practice, we emulate on an FPGA board [26] a System-on-Chip (SoC) including software. We use a Leon3-based SoC [24] as our hardware-prototype. This prototype is a full-blown SoC,

Table 1. TAB-BackSpace vs nuTAB-BackSpace Experiments. We use the same crash states. “# of Successful Iterations” is the number of iterations before timing out or reaching the set limit of 20. The timeout per iteration was chosen to be 5h. Each simulation run averages 10 minutes. “# of Chip Runs” is the total number of iterations plus the number of retries. Because “# of Chip Runs” is an aggregate, when the number of iterations for TAB is smaller than the number of iterations for nuTAB, the number of nuTAB runs may be greater than TAB runs (e.g., 1, 13-15). Crash states with a † are states that nuTAB-BackSpace computed all 20 iterations, but somewhere during the computation it deviated from the “expected” trace (in simulation, we can determine if the run reached the specified “crash” state). Therefore, these might be spurious traces. We suspect that, at some iteration, the normalized region of two different traces was too small to discriminate them.

Crash State	# of Successful Iterations		# of Chip Runs	
	TAB	nuTAB	TAB	nuTAB
1	0	11	62	143
†2	0	20	339	21
3	0	20	67	52
4	0	20	77	75
5	0	20	79	24
6	0	20	93	27
7	0	20	283	28
†8	0	20	128	20
†9	0	20	58	23
10	0	20	342	20
11	1	20	173	57
12	2	7	204	137
13	3	15	399	536
14	3	20	134	144
15	4	19	270	661

Crash State	# of Successful Iterations		# of Chip Runs	
	TAB	nuTAB	TAB	nuTAB
†16	4	20	112	27
17	5	20	157	28
18	5	20	199	41
19	6	6	120	58
†20	6	20	60	26
21	6	20	181	24
22	6	20	788	24
†23	7	20	568	22
24	12	20	251	27
25	12	20	308	25
26	15	20	534	20
27	20	20	282	28
28	20	20	403	29
29	20	20	463	52
30	20	20	726	26

with a SPARC V8 compatible core, AMBA bus, video, DDR2, Ethernet, i2c, and keyboard and mouse controllers. This SoC also has built-in debug features that can be enabled. In particular, we enable the provided trace buffer, LOGAN, but with a minimal configuration. The LOGAN has no signal compression. The signals we monitor are a combination of AMBA bus signals and some signals of the SPARC V8’s execution-pipeline-stage, totaling 134 signals.

Since one of the goals of demonstrating nuTAB-BackSpace on a hardware-prototype is to show that it works in a real (or as realistic as possible) debugging environment, we run non-trivial software on the Leon3. In our experiments, we are booting Linux (Linux Kernel 2.6.21).

Our debug scenario is as follows: while booting Linux, we want to derive the sequence of CPU and bus operations leading to the kernel’s function *start_kernel*. Thus, *start_kernel* is our “crash” state. The boot sequence up to this “crash” state is more than 20 million cycles deep. Simulating it with a logic simulator is impractical given this depth. Similarly, model checking it is infeasible.

The first experiment is to try TAB-BackSpace. We follow the same steps as Algorithm 1. The main difference is that instead of normalizing the extracted trace, we try to find an exact match on the overlap region between the current and previous traces. We set an address within *start_kernel* function as our breakpoint and run the chip; when it breakpoints, we extract a trace. From that trace, we pick a trace-buffer entry as our new crash-state and repeat. In our experiments, we set 2 hours as our retry timeout limit. The result of these experiments is a total of 207 chip runs, all of which breakpoint successfully, but none overlap cycle-by-cycle. In other words, we cannot TAB-BackSpace at all because, at each run, non-determinism changes the path the chip takes and so the probability of an exact match is too low.

The next experiment is to try nuTAB-BackSpace using the same scenario as before. However, we need to define the rewrite rules first. In this case, the SoC was built entirely from third-party IP, so our learning process was from the documentation and trace buffer dumps from the actual system running. Studying the trace buffer dumps, we observed that sometimes entire trace-buffers might have not a single video-controller transaction. Also, we noted that nullified instructions, although they vary from run to run, do not affect overall functionality of a system run. Therefore, for this debug scenario, we hypothesize that traces may have video-controller activity occurring at essentially arbitrary times, and that nullified instructions can be ignored. From our understanding of the design, we can create rewrite-rules easily to formalize the hypotheses and test them. (If our hypotheses produced uninterested traces, we would start again with a new hypothesis, creating new rewrite rules to try.)

We define the rewrite rules using the same notation as we used for the Router. Let $\text{Proj}(\cdot)_{ahbm}$ and $\text{Proj}(\cdot)_{inst}$ be two projection functions that map abstract-states, a , onto the subset of AMBA signals, which identify the current bus-master and onto the subset of signals from the CPU that define whether an instruction has been nullified. We can define R as follows:

$$\forall a. \text{Proj}(a)_{ahbm} = 0x3 . a \rightarrow \epsilon \quad (2)$$

$$\forall a. \text{Proj}(a)_{inst} = annul . a \rightarrow \epsilon \quad (3)$$

The rewrite rules ignore AMBA bus transactions from the video-controller and states where instructions have been nullified in the CPU's execution pipeline stage. (Note that the ignored cycles do not get deleted from the generated trace — the rewriting is solely to establish equivalence on the overlap region. The generated trace will always consist of actual states taken from trace buffer dumps.)

As in Subsection 4.1, we need to show that $\text{Leon3}_{RS}(\Sigma^*, R)$ is Noetherian, confluent, and concretization-preserving. As before, the system is length-reducing, and hence Noetherian. No two rules have an overlapping left-hand side. Consequently, there are no critical pairs, so $\text{Leon3}_{RS}(\Sigma^*, R)$ is locally confluent. The argument for concretization preservation is again based on insight into the design. The video controller bus transactions are irrelevant to the boot sequence

Table 2. nuTAB-BackSpace on Leon3. *Trace-Buffer Length* is the physical depth of the trace-buffer. Since we do not use compression, its depth is fixed. *Normalization-Region Length* is the number of cycles in the current trace-buffer that we normalize and use as a reference for the next trace-buffer. *New Cycles* is the number of new states present in the current trace-buffer.

Trace #	Trace-Buffer Length	Normalization Region Length	Normalized Length	New Cycles	Accumulated new cycles
1	1024	904	354	1024	1024
2	1024	519	137	384	1408
3	1024	781	133	241	1649
4	1024	680	168	514	2163
5	1024	709	168	348	2511
6	1024	892	141	45	2556
7	1024	—	—	398	2954

and can be arbitrarily ignored.² Similarly, nullified instructions have no effect on the (bus-level) debugging process, so they can be safely ignored as well. Any concrete execution trace which has these ignorable states corresponds to a concrete execution trace where those states have been deleted.

We show the results in Table 2. We iterated 7 times, resulting in a trace more than 2.5x the length of a single trace-buffer. Unlike TAB-BackSpace, the new technique handles the non-determinism, computing an abstract trace based on the trace-buffer signals.

5 Conclusion and Future Work

We have presented nuTAB-BackSpace, a novel technique to compute post-silicon debug traces in the presence of non-determinism. We exploit the observation that traces that are not cycle-by-cycle equal still share similarities from the debug engineer’s point-of-view. We let the user provide rewrite rules, and under some reasonable assumptions, we prove that nuTAB-BackSpace computes an abstract trace that concretizes to a trace that is reachable and leads to the crash state. We have demonstrated the effectiveness of nuTAB-BackSpace both in simulation and in hardware, computing abstract traces even when TAB-BackSpace cannot.

Increasingly, complex chips have many clock-domains and even completely asynchronous domains. Capturing traces on these designs and reasoning about them is a major challenge. We believe nuTAB-BackSpace holds promise for this problem, and this is the direct line of future work.

² Technically, ignoring video controller transactions is not truly concretization preserving, since any real concrete trace *will* have the occasional video transaction, whose timing is determined by state hidden in the video controller and the external video hardware. What the rewrite rule is really specifying is that that hidden state is irrelevant for the current debugging scenario. If we were debugging some video controller timing interaction, we would use different rewrite rules.

References

1. Abramovici, M., Bradley, P., Dwarakanath, K., Levin, P., Memmi, G., Miller, D.: A Reconfigurable Design-for-Debug Infrastructure for SoCs. In: DAC. IEEE (2006)
2. Adir, A., Golubev, M., Landa, S., Nahir, A., Shurek, G., Sokhin, V., Ziv, A.: Threadmill: a post-silicon exerciser for multi-threaded processors. In: DAC. IEEE (2011)
3. Anis, E., Nicolici, N.: Low Cost Debug Architecture using Lossy Compression for Silicon Debug. In: DATE. IEEE (2007)
4. ARM. Embedded Trace Macrocell Architecture Specification. Trace and Debug. ARM (2007), Ref: IH00140
5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge Univ. Press (1998)
6. Basu, K., Mishra, P., Patra, P.: Efficient combination of trace and scan signals for post silicon validation and debug. In: International Test Conference (ITC 2011). IEEE (2011)
7. Bentley, B., Gray, R.: Validating the Intel Pentium 4 processor. Intel Technology Journal (Quarter 1, 2001)
8. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
9. Book, R.V., Otto, F.: String-Rewriting Systems. Springer (1993)
10. Chang, K.H., Markov, I.L., Bertacco, V.: Automating Post-Silicon Debugging and Repair. In: ICCAD (2007)
11. Chang, K.H., Bertacco, V., Markov, I.L.: Simulation-Based Bug Trace Minimization With BMC-Based Refinement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26(1), 152–165 (2007)
12. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM TOPLAS 8(2), 244–263 (1986)
13. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. In: POPL, pp. 343–354 (1992)
14. de Paula, F.M., Gort, M., Hu, A.J., Wilton, S.J.E., Yang, J.: BackSpace: formal analysis for post-silicon debug. In: FMCAD. IEEE (2008)
15. de Paula, F.M., Nahir, A., Nevo, Z., Orni, A., Hu, A.J.: TAB-BackSpace: Unlimited-length trace buffers with zero additional on-chip overhead. In: DAC. IEEE (2011)
16. Hong, T., Li, Y., Park, S.B., Mui, D., Lin, D., Kaleq, Z.A., Hakim, N., Naeimi, H., Gardner, D.S., Mitra, S.: QED: Quick Error Detection tests for effective post-silicon validation. In: International Test Conference (ITC). IEEE (2010)
17. Klug, H.P.: Microprocessor testing by instruction sequences derived from random patterns. In: International Test Conference (ITC). IEEE (1988)
18. Ko, H.F., Nicolici, N.: Algorithms for State Restoration and Trace-Signal Selection for Data Acquisition in Silicon Debug. IEEE TCAD 28(2), 285–297 (2009)
19. Lee, D.H., Reddy, S.M.: On Determining Scan Flip-Flops in Partial-Scan Designs. In: IEEE International Computer-Aided Design. Digest of Technical Papers, pp. 322–325. IEEE International (November 1990)
20. Park, S., Yang, S., Cho, S.: Optimal State Assignment Technique for Partial Scan Designs. Electronics Letters 36(18), 1527–1529 (2000)

21. Park, S.B., Mitra, S.: IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors. In: DAC. IEEE (2008)
22. Prabhakar, S., Hsiao, M.: Using Non-trivial Logic Implications for Trace Buffer-Based Silicon Debug. In: Asian Test Symposium. IEEE (2009)
23. Quinton, B.R., Wilton, S.J.E.: Concentrator Access Networks for Programmable Logic Cores on SoCs. In: Int'l. Symp. on Circuits and Systems. IEEE (2005)
24. Web Reference, <http://www.gaisler.com>
25. Web Reference, <http://www.opencores.org>
26. Web Reference, <http://www.xilinx.com/univ/xupv5-1x110t.html>
27. Riley, M., Chelstrom, N., Genden, M., Sawamura, S.: Debug of the CELL Processor: Moving the Lab into Silicon. In: International Test Conference. IEEE (2006)
28. Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H., Sakallah, K.A.: Improved Design Debugging Using Maximum Satisfiability. In: Formal Methods in Computer-Aided Design. IEEE (2007)
29. Williams, M.J.Y., Angell, J.B.: Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic. IEEE TC C-22(1), 46–60 (1973)
30. Yang, J.S., Touba, N.A.: Expanding Trace Buffer Observation Window for In-System Silicon Debug through Selective Capture. In: VLSI Test Symposium 2008. IEEE (2008)
31. Zhu, C.S., Weissenbacher, G., Malik, S.: Post-silicon fault localisation using maximum satisfiability and backbones. In: Formal Methods in Computer-Aided Design (FMCAD). IEEE (2011)

Incremental, Inductive CTL Model Checking*

Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi

ECEE Department, University of Colorado at Boulder
`{zyad.hassan, bradleya, fabio}@colorado.edu`

Abstract. A SAT-based incremental, inductive algorithm for model checking CTL properties is proposed. As in classic CTL model checking, the parse graph of the property shapes the analysis. However, in the proposed algorithm, called IICTL, the analysis is directed by task states that are pushed down the parse tree. To each node is associated over- and under-approximations to the set of states satisfying that node’s property; these approximations are refined until a proof that the property does or does not hold is obtained. Each CTL operator corresponds naturally to an incremental sub-query: given a task state, an **EX** node executes a SAT query; an **EU** node applies IC3; and an **EG** node applies FAIR. In each case, the query result provides more general information than necessary to satisfy the task. When a query is satisfiable, the returned trace is generalized using forall-exists reasoning, during which IC3 is applied to obtain new reachability information that enables greater generalization. When a query is unsatisfiable, the proof provides the generalization. In this way, property-directed abstraction is achieved.

1 Introduction

Incremental, inductive verification (IIV) algorithms construct proofs by generating lemmas based on concrete hypothesis states. Through inductive generalization, a lemma typically provides significantly more information than is required to address the hypotheses. A principle of IIV is that each lemma holds *relative to* previously generated lemmas, hence the term *incremental*, so that the difficulty of lemma generation is fairly uniform throughout execution. In this way, property-directed abstraction is achieved. The safety model checker IC3 [3, 4] and the model checker FAIR [6] for analyzing ω -regular properties are both incremental, inductive model checkers. IC3 generates stepwise relatively inductive clauses in response to states that lead to property violations. FAIR generates inductive information about reachability and SCC-closed sets in response to sets of states that together satisfy every fairness constraint. This paper describes an incremental, inductive model checker, IICTL, for analyzing CTL properties of finite state systems, possibly with fairness constraints.

An investigation into an IIV model checker for CTL properties is important for several reasons. First, CTL is a historically significant specification language. Second, some properties like *resetability* ($\text{AG } \text{EF } p$ in CTL) require branching time

* Work supported in part by the Semiconductor Research Corporation under contracts GRC 1859 and GRC 2220.

semantics. Third, on properties in the fragment common to CTL and LTL, traditional CTL algorithms are sometimes superior to traditional LTL algorithms. CTL model checking is inherently hierarchical in that a CTL property can be analyzed according to its parse DAG. In the context of IIV, the strategy that IICTL applies to such properties is different than that applied by FAIR. Finally, CTL offers a conceptual challenge that previous IIV algorithms, IC3 and FAIR, do not address: branching time semantics. In particular, CTL motivates generalizing counterexample traces in addition to using proof-based generalization.

IICTL builds on traditional parse DAG-based analyses, except that it eschews the standard global, or bottom-up, approach in favor of a task-directed strategy. Beginning with the initial states for the root node, task states—which, as in previous IIV algorithms, are concrete states of the system—are pushed down the DAG, directing a node to *decide* whether those states satisfy its associated subformula. In the process of making a decision, a node can in turn generate a set of tasks for its children, and so on. Depending on the root operator of the node, it applies a SAT solver (EX), a safety model checker such as IC3 (EU), or a fair cycle finder such as FAIR (EG), to investigate the status of the task states. Once it reaches a conclusion, it generalizes the witness—either a proof or a counterexample trace—to provide as much new information as possible.

The first approaches to SAT-based CTL model checking [1, 13] were global algorithms that leveraged the ability of CNF formulae and Boolean circuits to be reasonably sized in some cases when BDDs are not. They differ from IICTL, which is an incremental, local algorithm. McMillan [13] first proposed an efficient technique for quantifier elimination that is related to the algorithm of Section 3.2, but is not driven by a trace to be generalized. The idea of creating an unsatisfiable query to generalize a solution to a satisfiable one (used in (15) and (20) in Section 3.2) comes from [16] and is present also in [7]. A few attempts [20, 19, 14] have been made to extend bounded model checking to branching time. They are all restricted to universal properties, though, and they have not received an extensive experimental evaluation. Their effectiveness thus remains unclear.

After preliminaries in Section 2, Section 3 describes IICTL in detail. Section 4 presents the results of a prototype implementation of IICTL within the llmc model checker [11].

2 Preliminaries

A *finite-state system* is represented as a tuple $S : (\bar{i}, \bar{x}, I(\bar{x}), T(\bar{x}, \bar{i}, \bar{x}'), \mathcal{B})$ consisting of primary inputs \bar{i} , state variables \bar{x} , a propositional formula $I(\bar{x})$ describing the initial configurations of the system, a propositional formula $T(\bar{x}, \bar{i}, \bar{x}')$ describing the transition relation, and a set $\mathcal{B} = \{B_1(\bar{x}), \dots, B_\ell(\bar{x})\}$ of Büchi fairness constraints.

Primed state variables \bar{x}' represent the next state. A state of the system is an assignment of Boolean values to all variables \bar{x} and is described by a *cube* over \bar{x} , which, generally, is a conjunction of literals, each *literal* a variable or its negation. An assignment s to all variables of a formula F either satisfies the formula, $s \models F$, or falsifies it, $s \not\models F$. If s is interpreted as a state and $s \models F$,

then s is an *F-state*. A formula F implies another formula G , written $F \Rightarrow G$, if every satisfying assignment of F satisfies G .

The transition structure is assumed to be complete. That is, every state has at least one successor on every input: $\forall \bar{x}, \bar{i}. \exists \bar{x}' . (\bar{x}, \bar{i}, \bar{x}') \models T$. A *path* in S , s_0, s_1, s_2, \dots , which may be finite or infinite in length, is a sequence of states such that for each adjacent pair (s_i, s_{i+1}) in the sequence, $\exists \bar{i}. (s_i, \bar{i}, s'_{i+1}) \models T$. If $s_0 \models I$, then the path is a *run* of S . A state that appears in some run of the system is *reachable*. A path s_0, s_1, s_2, \dots is *fair* if, for every $B \in \mathcal{B}$, infinitely many s_i satisfy B , $s_i \models B$; if $s_0 \models I$ then it is a *fair run* or *computation* of S .

Computational Tree Logic (CTL [8, 15]) is a branching-time temporal logic. Its formulae are inductively defined over a set A of atomic propositions. Every atomic proposition is a formula. In addition, if φ and ψ are CTL formulae, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\text{EX } \varphi$, $\text{E } \psi \cup \varphi$, and $\text{EG } \varphi$. Additional operators are defined as abbreviations. In particular, $\text{EF } \varphi$ abbreviates $\text{E}(\varphi \vee \neg\varphi) \cup \varphi$, $\text{AX } \varphi$ abbreviates $\neg \text{EX } \neg\varphi$, $\text{AG } \varphi$ abbreviates $\neg \text{EF } \neg\varphi$, and $\text{AF } \varphi$ abbreviates $\neg \text{EG } \neg\varphi$. A model of a CTL formula is a pair $M = (S, \mathcal{V})$ of a finite-state system S and a valuation \mathcal{V} of the atomic propositions as subsets of states of S . Satisfaction of a CTL formula at state s_0 of M is then defined as follows:

$M, s_0 \models a$	iff $s_0 \in \mathcal{V}(a)$ for $a \in A$
$M, s_0 \models \neg\varphi$	iff $M, s_0 \not\models \varphi$
$M, s_0 \models \varphi \wedge \psi$	iff $M, s_0 \models \varphi$ and $M, s_0 \models \psi$
$M, s_0 \models \text{EX } \varphi$	iff \exists a fair path s_0, s_1, \dots of S such that $M, s_1 \models \varphi$
$M, s_0 \models \text{EG } \varphi$	iff \exists a fair path s_0, s_1, \dots of S such that for $i \geq 0$, $M, s_i \models \varphi$
$M, s_0 \models \text{E } \psi \cup \varphi$	iff \exists a fair path s_0, s_1, \dots of S such that there exists $i \geq 0$ for which $M, s_i \models \varphi$, and for $0 \leq j < i$, $M, s_j \models \psi$.

Then $M \models \varphi$ if $\forall s . s \models I \Rightarrow M, s \models \varphi$. That is, M models formula φ if all its initial states do. In model M , the set of states that satisfy φ is written $[\![\varphi]\!]$.

That every CTL formula is interpreted as a set of states makes model checking easier than for the more expressive CTL*. Working bottom-up on the parse graph of φ , the standard symbolic CTL model checking algorithm [12] annotates each node with a set of states. Boolean connectives are dealt with in the obvious way, while temporal operators are handled with fixpoint computations. The bottom-up approach is also known as *global* model checking. In contrast, *local* model checking [10, 17, 2, 9] proceeds top-down. A local model checker starts from the goal of proving that initial state s satisfies φ and applies inference rules to reformulate the goal as a list of subgoals in terms of subformulae of φ and states in the vicinity of s . While local model checking can sometimes prove a property without examining most of a system's states, in its basic formulation it does not play to the strengths of symbolic algorithms. For that reason, local model checkers for finite-state systems tend to employ explicit search [1].

¹ Some BDD-based model checkers incorporate elements of local algorithms. For instance, the CTL model checker in VIS [18] uses top-down *early termination conditions* to define conditions that a safe approximation of a set of states must satisfy. However, it is still fundamentally a bottom-up algorithm.

Table 1. Initial bounds for IICTL

ψ_v	L_v	U_v	ψ_v	L_v	U_v
$a \in A$	$\mathcal{V}(a)$	$\mathcal{V}(a)$	$\text{EX } \psi_i$	\perp	\top
$\neg\psi_i$	$\neg U_i$	$\neg L_i$	$\text{E } \psi_j \text{ U } \psi_i$	L_i	$U_i \vee U_j$
$\psi_i \wedge \psi_j$	$L_i \wedge L_j$	$U_i \wedge U_j$	$\text{EG } \psi_i$	\perp	U_i

3 Algorithm

The input to IICTL consists of a model $M = (S, \mathcal{V})$ and the parse graph of a CTL formula φ . Each node of the parse graph is a natural number v and is labeled with a token from φ . Node 0 is the root of the DAG. The formula rooted at v is denoted by ψ_v , so that, in particular, $\psi_0 = \varphi$. IICTL annotates each node v with two propositional formulae over the state variables: U_v and L_v , with which an upper bound formula \mathcal{U}_v and a lower bound formula \mathcal{L}_v , discussed later, approximate the satisfying set $\llbracket \psi_v \rrbracket$ of the formula ψ_v in M . Initial approximations are computed as shown in Table 1. A global approximation of the states of S reachable from the initial states is maintained as inductive propositional formula R . Initially, $R = \top$; that is, all states are presumed reachable.

Throughout execution, IICTL maintains the following invariant:

$$\llbracket R \wedge U_v \wedge L_v \rrbracket \subseteq \llbracket R \wedge \psi_v \rrbracket \subseteq \llbracket R \wedge U_v \rrbracket . \quad (1)$$

All states of the left set definitely satisfy ψ_v ; all states not in the right set definitely do not satisfy ψ_v or are unreachable. A state s of the system S such that $s \models R \wedge U_v$ but $s \not\models R \wedge U_v \wedge L_v$ —together, $s \models R \wedge U_v \wedge \neg L_v$ —is *undecided* for ψ_v . The algorithm incrementally refines the approximations by considering undecided states until either no initial state of S is undecided for φ , proving $M \models \varphi$, or an initial state \hat{s} is found such that $\hat{s} \not\models U_0$, proving $M \not\models \varphi$.

Let $\mathcal{L}_v = R \wedge U_v \wedge L_v$ designate the *lower bound* states: those states that are known to satisfy ψ_v . Let $\mathcal{U}_v = R \wedge U_v$ designate the *upper bounds* states: those states that are not known not to satisfy ψ_v . Invariant (1) is then written $\llbracket \mathcal{L}_v \rrbracket \subseteq \llbracket R \wedge \psi_v \rrbracket \subseteq \llbracket \mathcal{U}_v \rrbracket$. Finally, let $\mathcal{A}_v = \mathcal{U}_v \wedge \neg \mathcal{L}_v = R \wedge U_v \wedge \neg L_v$ designate the undecided states of node v .

Section 3.1 describes the essential structure of IICTL in detail. Section 3.2 introduces *forall-exists generalization*, which is applied to counterexample traces. Then Section 3.3 describes two important refinements to the basic algorithm, while Section 3.4 describes the additions for handling fairness constraints.

3.1 An Outline of IICTL

If ever $I \wedge \neg U_0$ becomes satisfiable, then IICTL concludes that $M \not\models \varphi$: not even the over-approximation \mathcal{U}_0 of φ contains all I -states, so neither can φ itself. If instead $I \wedge \neg(L_0 \wedge U_0)$ becomes unsatisfiable, then $M \models \varphi$: the under-approximation \mathcal{L}_0 of φ contains all I -states, so φ itself must as well.

Otherwise, one or more initial undecided states must be *decided*. At the top level, a witness s to the satisfiability of $I \wedge U_0 \wedge \neg L_0$ is undecided; it is decided by calling the recursive function `decide` with arguments s and 0, the root of the parse tree of φ , which eventually returns `true` if $M, s \models \varphi$ and `false` otherwise. In general, `decide(t, v)` return `true` iff $M, t \models \psi_v$. A call to `decide(t, v)` can update L_v or U_v (or both) so that state t becomes decided for ψ_v ; moreover, the call can trigger a cascade of recursive calls that update the bounds of descendants of v and, crucially, may decide many states besides t . Within the tree, the over-approximating reachability set R becomes relevant: a state t is undecided at node v if it satisfies \mathcal{A}_v . The pseudocode for `decide` listed in Figure 1 provides structure to the following discussion.

Boolean Nodes. According to Table 1, no state can be undecided for a propositional node because the initial approximations are exact; therefore, in the case that v is a propositional node, one of the conditions of lines 2–3 holds.

If $\psi_v = \psi_u \wedge \psi_w$, the following invariant is maintained:

$$U_v = U_u \wedge U_w \quad \text{and} \quad L_v = L_u \wedge L_w . \quad (2)$$

If t is undecided at entry, then recurring on nodes u and w decides t for v (line 6). The `update` statement (line 5; also lines 7, 20, and 32) indicates that L_v and U_v should be updated whenever a child’s bound is updated during recursion. It does not express an invariant.

If $\psi_v = \neg\psi_u$, the following invariant is maintained:

$$U_v = \neg(L_u \wedge U_u) \quad \text{and} \quad L_v = \neg U_u . \quad (3)$$

If t is undecided at entry, then recurring on node u decides t for v (line 8).

EX Nodes. If $\psi_v = \text{EX } \psi_u$, then the undecided question is whether t has a successor satisfying ψ_u . ICTL executes two SAT queries in order to answer this question. First, it executes an *upper bound query*. Naively, this query is $t \wedge T \wedge U'_u$, which asks whether t has a U'_u -successor. However, for better generalization, the following is used instead (line 10):

$$t \wedge \mathcal{U}_v \wedge T \wedge U'_u . \quad (4)$$

If unsatisfiable, the core reveals cube $\bar{t} \subseteq t$ such that all \bar{t} -states (including t) lack U'_u -successors (and thus ψ_u -successors) or are unreachable. U_v is then updated to $U_v \wedge \neg \bar{t}$ (line 11)—no \bar{t} -state is a ψ_u -state (or it is an unreachable ψ_u -state).

However, if query 4 is satisfiable, the witness reveals successor U'_u -state s (line 13). A *lower bound query* is executed next (line 14):

$$t \wedge T \wedge L'_u \wedge U'_u . \quad (5)$$

If satisfiable, then t itself has been decided: it definitely has a ψ_u -successor, since it has a $(U'_u \wedge L'_u)$ -successor (recall invariant 1). Forall-exists generalization

```

1  bool decide(t : state, v : node):
2    if  $t \models R \wedge U_v \wedge L_v$ : return true {already decided:  $M, t \models \psi_v$ }
3    if  $t \not\models R \wedge U_v$ : return false {already decided:  $M, t \not\models \psi_v$ }
4    match  $\psi_v$  with:
5       $\psi_u \wedge \psi_w$ : [update  $L_v$ ,  $U_v := L_u \wedge L_w$ ,  $U_u \wedge U_w$ ]
6        return decide(t, u)  $\wedge$  decide(t, w)
7       $\neg\psi_u$ : [update  $L_v$ ,  $U_v := \neg U_u$ ,  $\neg(L_u \wedge U_u)$ ]
8        return  $\neg$ decide(t, u)
9    EX  $\psi_u$ :
10      if  $t \wedge U_v \wedge T \wedge U'_u$  is unsat: {with  $\bar{t} \subseteq t$  from core}
11         $U_v := U_v \wedge \neg \bar{t}$ 
12        return false
13      else: {with t-successor s}
14        if  $t \wedge T \wedge L'_u \wedge U'_u$  is sat:
15           $L_v := L_v \vee \text{generalize}(t)$ 
16          return true
17        else:
18          decide(s, u)
19          return decide(t, v)
20    E  $\psi_u \cup \psi_w$ : [update  $L_v$ ,  $U_v := L_v \vee L_w$ ,  $U_v \wedge (U_u \vee U_w)$ ]
21      if  $\neg(t \wedge U_w$  is sat or reach( $S, U_u \wedge U_v \wedge R \wedge U'_v, t, U_w$ ):  

22         $U_v := U_v \wedge \neg P$  {with proof P}
23        return false
24      else: {with trace  $s_0 = t, s_1, \dots, s_n$ }
25        if  $t \wedge L_w$  is sat or reach( $S, L_u \wedge U_v \wedge U'_v, t, L_v \wedge U_v$ ):
26           $L_v := L_v \vee \text{generalize}(\bar{r})$  {with trace  $\bar{r}$ }
27          return true
28        elif decide( $s_i, u$ ),  $0 \leq i < n$ , and decide( $s_n, w$ ) are true:
29           $L_v := L_v \vee \text{generalize}(s_0, \dots, s_n)$ 
30          return true
31        else: return decide(t, v)
32    EG  $\psi_u$ : [update  $U_v := U_v \wedge U_u$ ]
33      if  $\neg \text{fair}(S, U_v \wedge R \wedge U'_v, t)$ : {with assertion P}
34         $U_v := U_v \wedge \neg P$ 
35        return false
36      else: {with trace  $s_0 = t, \dots, s_k, \dots, s_n, s_k$ }
37        if  $\text{fair}(S, L_u \wedge U_v \wedge R \wedge L'_u \wedge U'_v, t)$ : {with trace  $\bar{r}$ }
38           $L_v := L_v \vee \text{generalize}(\bar{r})$ 
39          return true
40        elif decide( $s_i, u$ ),  $0 \leq i \leq n$ , are true:
41           $L_v := L_v \vee \text{generalize}(s_0, \dots, s_n)$ 
42          return true
43        else: return decide(t, v)

```

Fig. 1. Basic version of the main recursive function

(Section B.2) then produces a cube $\bar{t} \subseteq t$ of states that definitely have ψ_u -successors (or are unreachable), and L_v is updated to $L_v \vee \bar{t}$ (line 15). If the query is unsatisfiable (line 17), then apparently state s is undecided for u . In this case, $\text{decide}(s, u)$ is called (line 18), which results in updates to at least one

of U_u and L_u , which is a form of progress. The entire process iterates until t is decided (line 19).

EU Nodes. An EU-node maintains the following invariant:

$$\llbracket \mathcal{L}_w \rrbracket \subseteq \llbracket \mathcal{L}_v \rrbracket, \llbracket \mathcal{L}_v \rrbracket \subseteq \llbracket \mathcal{L}_u \rrbracket \cup \llbracket \mathcal{L}_w \rrbracket, \llbracket \mathcal{U}_w \rrbracket \subseteq \llbracket \mathcal{U}_v \rrbracket, \llbracket \mathcal{U}_v \rrbracket \subseteq \llbracket \mathcal{U}_u \rrbracket \cup \llbracket \mathcal{U}_w \rrbracket. \quad (6)$$

If $\psi_v = \mathsf{E} \psi_u \mathsf{U} \psi_w$, then the undecided question is whether t has a ψ_u -path to a ψ_w -state. To answer this question, it executes two reachability queries using an engine capable of returning counterexample traces and inductive proofs, such as IC3 [3, 4]. Let $\text{reach}(S, C, F, G)$ be a function that accepts a system S , a set of constraints $C(\bar{x}, \bar{x}')$ on the transition relation, an initial condition F , and a target G ; and that returns either a counterexample run from an F -state to a G -state, or an assertion $P(x)$, inductive relative to C , separating F from G .

The *upper bound query* asks whether t leads to a U_w -state (line 21). First, the query $t \wedge U_w$ determines if t is itself a U_w -state. If not, then the following query determines if it can reach a U_w -state via a U_u -path:

$$\text{reach}(S, U_u \wedge U_v \wedge R \wedge U'_v, t, U_w) . \quad (7)$$

The transition relation constraint $U_u \wedge U_v \wedge R \wedge U'_v$ mixes the necessary (U_u) with the optimizing ($U_v \wedge R \wedge U'_v$). If the query is unsatisfiable, the returned inductive proof P shows that no U_w -state can be reached via a potentially reachable $U_u \wedge U_v$ -path, deciding at least t and informing the update of U_v to $U_v \wedge \neg P$ (line 22). If either of the query $t \wedge U_w$ or query (7) is satisfiable, let $s_0 = t, s_1, \dots, s_n$ be the returned counterexample trace (line 24).

Lower bound queries are executed next (line 25). First, *decide* asks if t is itself a ψ_w -state via the query $t \wedge \mathcal{L}_w$. If not, it asks whether t can reach a known ψ_v -state via a known ψ_u -path:

$$\text{reach}(S, \mathcal{L}_u \wedge U_v \wedge U'_v, t, L_v \wedge U_v) . \quad (8)$$

In this version, the target set has those states that are known to have ψ_u -paths to ψ_w -states. If either case is satisfiable, t is decided for v : it has a ψ_u -path to a ψ_w -state. Forall-exists generalization (Section 3.2) produces a set of states F , including t , that definitely have ψ_u -paths to ψ_w -states or are unreachable, with which L_v is updated (line 26).

However, if the query is unsatisfiable, then attention returns to the trace s_0, \dots, s_n of the upper bound query (7) to decide whether its states satisfy the appropriate subformulae (lines 28–31). Each s_i , $0 \leq i < n$, is queried for node u , and s_n is queried for node w . If all states of the trace² are decided positively (line 28), then t is decided positively for v ; therefore, L_v is expanded by the generalization of the trace (line 29). If one of the states is decided negatively, the upper and lower bound queries are iterated until t is decided (line 31): either a trace is found, or the nonexistence of such a trace is proved.

² Section 3.3 refines this state-by-state treatment to the level of traces.

EG Nodes. An EG-node maintains the following invariant:

$$\llbracket \mathcal{L}_v \rrbracket \subseteq \llbracket \mathcal{L}_u \rrbracket \quad \text{and} \quad \llbracket \mathcal{U}_v \rrbracket \subseteq \llbracket \mathcal{U}_u \rrbracket . \quad (9)$$

If $\psi_v = \text{EG } \psi_u$, then the undecided question is whether there exists a reachable fair cycle all of whose states are ψ_u -states. To answer this question, it executes two *fair cycle queries* using an engine capable of returning (1) fair cycles and (2) inductive reachability information describing states that lack a reachable fair cycle. FAIR is one such engine [6]. Let $\text{fair}(S, C, F)$ be a function that accepts a system S , possibly with fairness constraints $\{B_1, \dots, B_\ell\}$, a set of constraints $C(\bar{x}, \bar{x}')$ on the transition relation, and an initial condition F ; and that returns either an F -reachable fair cycle, or an inductive assertion P , where $F \Rightarrow P$, describing a set of states that lack reachable fair cycles.

The *upper bound query* asks whether a reachable fair cycle whose states satisfy U_v exists. The constraint on the transition relation uses U_v because states of a counterexample should potentially be $\text{EG } \psi_u$ states (line 33):

$$\text{fair}(S, U_v \wedge R \wedge U'_v, t) . \quad (10)$$

If the query is unsatisfiable, the returned inductive assertion P describes states, including t , that do not have reachable fair cycles (line 33); hence, U_v is updated to $U_v \wedge \neg P$ (line 34). Otherwise, a reachable fair cycle $s_0 = t, \dots, s_k, \dots, s_n, s_k$ is obtained (line 36).

Before exploring the trace, a *lower bound query* is executed (line 37) to determine whether a reachable fair \mathcal{L}_u -cycle exists³:

$$\text{fair}(S, \mathcal{L}_u \wedge \mathcal{L}'_u, t) . \quad (11)$$

If it is satisfiable, the resulting run is generalized (Section 3.2) to a formula F , and L_v is updated to $L_v \vee F$ (line 38).

Otherwise, the reachable fair cycle from query (10) is considered (line 40). If all s_i are ψ_u -states, `decide` finishes as with a satisfiable lower bound query (lines 41–42). Otherwise, the exploration updates U_v , so that some progress is made, and the process is iterated (line 43).

Even if `generalize` were to return what it is given, the sound updates to L_v (lines 15, 26, 29, 38, 41) and U_v (lines 11, 22, 34), combined with the progress guaranteed by each call to `decide`, make the basic version of IICCTL correct.

Theorem 1. *IICCTL terminates and returns `true` iff $M \models \varphi$.*

Example 1. Consider resetability, $\varphi = \text{AG EF } p = \neg \text{EF } \neg \text{EF } p$, whose parse graph, with initial upper and lower bounds is shown in Figure 2. Because initial states are undecided for 0, IICCTL chooses some initial state s and calls `decide`($s, 0$), which in turn calls `decide`($s, 1$). To determine if s is a ψ_1 -state, `decide` queries a safety model checker for the existence of a path from s to U_2 , i.e., to a $\neg p$ -state.

³ Note that the fair cycle query could potentially be avoided by asking if known ψ_v -states are reachable from t via a ψ_u -path: $\text{reach}(S, \mathcal{L}_u \wedge \mathcal{L}'_u, t, \mathcal{L}_v)$.

If none exist, inductive proof P is returned, and U_1 is updated by $\neg P$. If counterexample trace s, \dots, t is found, decide asks whether a path from s to L_2 exists, which is currently impossible. The disagreement between U_2 and L_2 on t triggers calls to $\text{decide}(t, 2)$ and $\text{decide}(t, 3)$. With equal bounds for node 4, only one reachability query is needed. If t cannot reach p (case 1), the inductive proof eliminates t from U_3 and adds it to L_2 . Then s can reach a ψ_2 -state, deciding s for 1 positively, and s, \dots, t are added to L_1 . Finally, s is removed from node 0, indicating failure of the property.

Otherwise (case 2), the discovered trace at node 3 is generalized to F , included in L_3 , and eliminated from U_2 . Then the upper bound reachability query of node 1 is repeated asking for the existence of a path from s to a $\neg p \wedge \neg F$ -state. The procedure continues until either case 1 occurs (failure), or until this query fails, establishing at least that $s \models \psi_0$. Then decide is invoked again for node 1 with a remaining undecided initial state if any exist, or success of the property is declared.

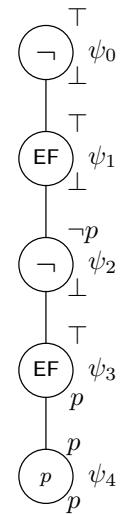


Fig. 2. AG EF p

3.2 Forall-Exists Generalization

Proofs from upper bound queries provide generalization in one direction: unsatisfiable cores for EX-nodes, inductive unreachability proofs for EU-nodes, and inductive reachability information from fair cycle queries for EG-nodes. While there are some techniques that ICTL applies to improve inductive proofs—proof strengthening, weakening, and shrinking—they have been discussed in the context of FAIR [6]. An essential aspect of making ICTL work in practice is the ability to generalize from counterexample traces. A first approach, given trace $s_0, i_0, s_1, i_1, \dots, s_{n-1}, i_{n-1}, s_n$ with interleaved states and primary input values, is to use the unsatisfiable cores of the query $s_j \wedge i_j \wedge T \wedge \neg s'_{j+1}$ to reduce s_j to a subcube, with decreasing j [16, 7]. For greater generalization power, *forall-exists generalization* is introduced in this section. While the overall idea is similar for the three operators EX, EU, and EG, details differ.

The overall idea of forall-exists trace generalization is to (1) select a cube c of the trace, (2) flip a literal of c to obtain \bar{c} , and (3) decide whether all $\mathcal{A}_v \wedge \bar{c}$ -states are ψ_v -states. If they are, c can be replaced with the resolvent of c and \bar{c} , that is, the cube obtained by dropping the literal of step (2). This process continues until no further literal of the trace can be dropped. During generalization, it is assumed that all states described by the current trace are L_v -states, so that an improvement of one cube can lead to improvements of other cubes. Hence, literals can be tried multiple times.

Selecting the cube (step (1)) and one of its literals (step (2)) can be heuristically guided. The following describes step (3) of the procedure, where \bar{c} is a candidate cube obtained by flipping a literal of a cube from the current trace.

EX Nodes. Let $\psi_v = \text{EX } \psi_u$, and let \bar{c} be a cube describing a set of states. If

$$\forall \bar{x}. \bar{c}(\bar{x}) \wedge \mathcal{A}_v(\bar{x}) \rightarrow \exists \bar{i}, \bar{x}' . T(\bar{x}, \bar{i}, \bar{x}') \wedge \mathcal{L}_u(\bar{x}') , \quad (12)$$

then \bar{c} can be added to L_v while maintaining invariant (II).

The challenge with (12) is quantifier alternation. Rather than using a general QBF solver, IICCTL adopts a strategy in which two queries are executed iteratively. The first SAT query is the following:

$$\bar{c} \wedge \mathcal{A}_v \wedge T \wedge \neg \mathcal{L}'_u . \quad (13)$$

It asks whether all successors of \bar{c} -states are \mathcal{L}_u states. If the query is unsatisfiable, then no undecided \bar{c} -state has a successor outside of \mathcal{L}_u . The states in $\bar{c} \wedge \neg \mathcal{A}_v$ can be added freely to L_v . Those in $\bar{c} \wedge \mathcal{A}_v$ have all their successors in \mathcal{L}_u ; hence, they satisfy ψ_v and can be added to L_v by updating it to $L_v \vee \bar{c}$.

If, however, query (13) is satisfiable, then there exists an undecided \bar{c} -state s with at least one successor outside of \mathcal{L}_u . The second SAT query establishes whether all of s 's successors are $\neg \mathcal{L}_u$ -states:

$$s \wedge T \wedge \mathcal{L}'_u . \quad (14)$$

If the query is satisfiable, then there exists s -successor state t and input j such that $t \models \mathcal{L}_u$ and $(s, j, t') \models T$. In this case, the following query is unsatisfiable:

$$s \wedge \mathcal{A}_v \wedge j \wedge T \wedge \neg \mathcal{L}'_u . \quad (15)$$

The set of literals s that do not appear in the unsatisfiable core can be dropped from s to obtain cube \bar{s} , which describes the set of states that either are not in \mathcal{A}_v or go to \mathcal{L}_u under input j ; these states can therefore be added to L_v , yielding $L_v \vee \bar{s}$. While query (15) is unsatisfiable even without the conjunction of \mathcal{A}_v , the presence of \mathcal{A}_v allows generalizations of s that include $\neg \mathcal{A}_v$ -states.

If query (14) is unsatisfiable, s is considered a *counterexample to generalization* (CTG). It explains why \bar{c} cannot be added to L_v at this time: it is known that s lacks an \mathcal{L}_u -successor and thus undecided whether s has a ψ_u -successor. Hence, s remains undecided for v . However, all is not lost: the question remains whether s is even reachable. Because generalization is unnecessary for correctness but necessary for (practical) completeness, answering this question requires balancing computational costs against the potential benefits of greater generalization. There are three reasonable approaches to addressing the question: (1) ignore it, obtaining immediate speed at the cost of generalization; (2) apply a semi-decision procedure for reachability, such as the MIC procedure of FSIS and IC3 [5, 4]; (3) apply a full reachability procedure such as IC3. In the latter two cases, proofs of unreachability refine R , which strengthens all IICCTL queries, in addition to making s irrelevant to the current generalization attempt.

With approach (3), even in the case that IC3 finds that s is reachable, the truly inductive clauses generated during the analysis are added to R , yielding new information. Furthermore, s is added to a set of states known to be reachable. Henceforth, whenever a cube \bar{c} is considered as part of generalization at

some node v , $s \in \bar{c}$ is first tested; if so, then query (14) is immediately applied. If this query is satisfiable, then s is marked as henceforth irrelevant for generalizations at node v . This reuse of known reachable states during generalization significantly mitigates the cost of approach (3) on some benchmarks.

EU and EG Nodes. Let node v be such that either $\psi_v = \text{E } \psi_u \text{ U } \psi_w$ or $\psi_v = \text{EG } \psi_u$. In both cases, the generalization queries are motivated by the following:

$$\forall \bar{x} . \bar{c}(\bar{x}) \wedge \mathcal{A}_v(\bar{x}) \rightarrow \mathcal{L}_u(\bar{x}) \wedge \exists \bar{i}, \bar{x}' . T(\bar{x}, \bar{i}, \bar{x}') \wedge \mathcal{L}_v(\bar{x}') . \quad (16)$$

Any $(\bar{c} \wedge \mathcal{A}_v)$ -state must be a ψ_u -state, motivating the $\mathcal{L}_u(\bar{x})$ term.⁴ Additionally, it must have a ψ_v -successor, motivating the $\mathcal{L}_v(\bar{x})$ term.

To address (16) without a QBF solver, several queries are executed iteratively. First, $\neg \psi_u$ -states are addressed with the following query:

$$\bar{c} \wedge \mathcal{A}_v \wedge \neg \mathcal{L}_u . \quad (17)$$

If satisfiable, the indicated CTG can be analyzed for reachability. A reachable CTG ends consideration of \bar{c} . Once query (17) becomes unsatisfiable, focus turns to the existence of ψ_v -successors for all relevant \bar{c} -states:

$$\bar{c} \wedge \mathcal{A}_v \wedge T \wedge \neg \mathcal{L}'_v . \quad (18)$$

If unsatisfiable, L_v is updated to $L_v \vee \bar{c}$, and generalization is complete.

Otherwise, a witness state s exists; it is checked for L_v -successors:

$$s \wedge T \wedge \mathcal{L}'_v . \quad (19)$$

If the query is satisfiable, then there exists s -successor state t and input j such that $t \models \mathcal{L}_v$ and $(s, j, t') \models T$. In this case, the following query is unsatisfiable:

$$s \wedge \mathcal{A}_v \wedge j \wedge T \wedge (\neg \mathcal{L}_u \vee \neg \mathcal{L}'_v) . \quad (20)$$

Its unsatisfiable core reveals a cube $\bar{s} \subseteq s$ with which to update L_v to $L_v \vee \bar{s}$, which eliminates s as a counterexample to query (18). If query (19) is unsatisfiable, then s is a CTG to be handled as described for EX nodes.

3.3 Refinements

Two refinements are immediate. First, to detect early termination, each time some node u 's L_u or U_u is updated, its parent v is notified, and the proper update is made to its L_v and U_v , as explained in Section 3.1. If there is a (semantic) change in at least one of L_v and U_v , then the upward propagation continues. If the root node is modified so that a termination criterion is met ($I \wedge \neg U_0$ is satisfiable or $I \wedge \neg L_0$ is unsatisfiable), then the proof is complete.

⁴ If v is an EU-node, notice that the term \mathcal{A}_v in the antecedent excludes considering \mathcal{L}_w -states of \bar{c} , for such states are already decided for node v .

Consider the property of Example 1. If it fails, there is at least one trace leading from an initial state to a state s that falsifies $\text{EF } p$. The outer EF node would direct ICTL to find such a trace, after which the upper bound query of the inner EF -node would return a proof that s cannot reach a p -state. As soon as the proof is generated, it is evident that the property is false.

Second, in Section 3.1, individual task states are submitted to nodes. However, multi-state initial conditions and counterexample traces create sets of task states. While generalization mitigates the cost of handling each state of a task set individually, even better is to allow nodes to reason about multi-state tasks. To do so, a node of the DAG receives a task set \mathcal{T} with an associated label. The label **All** indicates that all states $t \in \mathcal{T}$ must satisfy ψ_v , while the label **One** indicates that at least one state $t \in \mathcal{T}$ must satisfy ψ_v .

Now function `decide` takes three arguments: `decide(\mathcal{T}, ℓ, v)`, where $\ell \in \{\text{All}, \text{One}\}$. Initially, `decide($I, \text{All}, 0$)` is called. A general invocation `decide(\mathcal{T}, ℓ, v)` immediately returns `false` if $\ell = \text{All}$ and $\mathcal{T} \wedge \neg U_v$ is satisfiable, or if $\ell = \text{One}$ and $\mathcal{T} \wedge U_v$ is unsatisfiable; and returns `true` if $\ell = \text{One}$ and $\mathcal{T} \wedge \mathcal{L}_v$ is satisfiable, or if $\ell = \text{All}$ and $\mathcal{T} \wedge \neg \mathcal{L}_v$ is unsatisfiable. Otherwise, it updates \mathcal{T} to $\mathcal{T} \wedge \mathcal{A}_v$ —that is, the undecided subset of \mathcal{T} —and continues.

If v is a \neg -node, then it switches the label and passes the task onto its child. If v is a $\psi_u \wedge \psi_w$ -node and $\ell = \text{All}$, then `decide($\mathcal{T}, \text{All}, u$)` is called. A return value of `false` indicates that some state $t \in \mathcal{T}$ falsifies ψ_u , so this call returns `false` as well. Otherwise, `decide($\mathcal{T}, \text{All}, w$)` is called and its return value returned.

If $\ell = \text{One}$, the situation is more interesting: a state $t \in \mathcal{T}$ must be identified that satisfies both ψ_u and ψ_w . Therefore, `decide($\mathcal{T}, \text{One}, u$)` is called. A return value of `false` indicates that all states of \mathcal{T} falsify ψ_u , so this call returns `false`, too. However, a return value of `true` indicates that at least one state of \mathcal{T} satisfies ψ_u , and these states are now included in L_u . Therefore, `decide($\mathcal{T} \wedge \mathcal{L}_u, \text{One}, w$)` is called to see if any of the identified states also satisfies ψ_w . If so, this call returns `true`. If not, then v 's new bounds exclude some states of \mathcal{T} , including the ones that were found to satisfy ψ_u . \mathcal{T} is consequently set to $\mathcal{T} \wedge U_v \wedge \neg L_u$, and the process is iterated.

If v is an EX-, EU-, or EG-node and $\ell = \text{One}$, then its queries are executed iteratively with $\mathcal{T} \wedge U_v$ as the source states until either $\mathcal{T} \wedge \mathcal{L}_v$ is satisfiable, in which case `true` is returned, or $\mathcal{T} \wedge U_v$ is unsatisfiable, in which case `false` is returned. If $\ell = \text{All}$, then v 's queries are executed iteratively with $\mathcal{T} \wedge \neg \mathcal{L}_v$ as the source states until either $\mathcal{T} \wedge \neg U_v$ becomes satisfiable, in which case `false` is returned, or $\mathcal{T} \wedge \neg \mathcal{L}_v$ becomes unsatisfiable, in which case `true` is returned. The handling of multiple initial states by the `reach` and `fair` queries themselves is the main advantage of the multi-state refinement of `decide`.

With the handling of multi-state tasks defined, it remains to define how to create such tasks. Suppose during analysis of node v , where $\psi_v = \text{E } \psi_u \cup \psi_w$, `decide` discovers a trace s_0, \dots, s_n in which it must be decided whether states s_0, \dots, s_{n-1} are ψ_u -states and state s_n is a ψ_w -state. Then `decide($\{s_0, \dots, s_{n-1}\}, \text{All}, u$)` and `decide($\{s_n\}, \text{All}, w$)` are called. Similarly, if $\psi_v = \text{EG } \psi_u$, the states of a purported fair cycle s_0, \dots, s_n are decided with `decide($\{s_0, \dots, s_n\}, \text{All}, u$)`.

3.4 Fairness

Fairness in CTL cannot be handled completely within the logic itself. Instead, model checkers must be able to handle fairness constraints algorithmically when deciding whether a state satisfies an **EG** formula, a task that IICTL accomplishes by passing the constraints to **fair**. To show that finite paths computed for other types of formulae can be extended to fair paths, it suffices to show that they end in states that satisfy **EG T**. Hence, it is customary in BDD-based CTL model checkers to pre-compute the states that satisfy **EG T** and constrain the targets of **EU** and **EX** computations to them [2].

IICTL instead tries to decide the fairness of as few states as possible. To that effect, it computes from the given φ a modified formula $\tau(\varphi)$ recursively defined as follows:

$$\begin{array}{ll} \tau(p) = p & \tau(\text{EG } \varphi) = \text{EG } \tau(\varphi) \\ \tau(\neg\varphi) = \neg\tau(\varphi) & \tau(\text{EX } \varphi) = \text{EX}(\tau(\varphi) \wedge \psi) \\ \tau(\varphi_1 \wedge \varphi_2) = \tau(\varphi_1) \wedge \tau(\varphi_2) & \tau(\text{E } \varphi_1 \cup \varphi_2) = \text{E } \tau(\varphi_1) \cup (\tau(\varphi_2) \wedge \psi) , \end{array}$$

where

- p is an atomic proposition, and
- $\psi = \top$ if φ is a positive Boolean combination of **EX**, **EU** and **EG** formulae;
 $\psi = \text{EG } \top$ otherwise.

For example, $\tau(\text{AG EF}(p \wedge \neg q)) = \tau(\neg \text{EF} \neg \text{EF}(p \wedge \neg q)) = \neg \text{EF}(\neg \text{EF}((p \wedge \neg q) \wedge \text{EG } \top) \wedge \text{EG } \top)$, while $\tau(\text{AG AF } p) = \tau(\neg \text{EF EG } \neg p) = \neg \text{EF EG } \neg p$.

While the definition of $\tau(\varphi)$ is closely related to the one implicitly used by most BDD-based model checkers—the difference is that in the latter, ψ always equals **EG T**—it minimizes checks for fairness by taking into account that every path with a fair suffix is fair.

For instance, in the case of **AG AF p**, IICTL does not check whether any state satisfies **EG T** because the states that satisfy **EG p** are known to be fair. For the resetability property **AG EF**($p \wedge \neg q$), however, a state that satisfies $p \wedge \neg q$ is not assumed to satisfy the inner **EF** node unless it is proved fair.

4 Results

The IICTL algorithm has been implemented in the **Ilmc** model checker [1], and it has been evaluated on a set of 33 models (mostly from [18]) for a total of 363 CTL properties (278 passing and 85 failing). These properties include only one invariant, since IICTL delegates invariant checking to IC3. No collection of branching time properties used in real designs similar to that available for safety properties is available. This experimental evaluation is therefore preliminary. The experiments have been run on machines with 2.8 GHz Intel Core i7 CPUs and 9 GB of RAM. A timeout of 300 s was imposed on all runs.

In this section the performance of IICTL is compared to that of the BDD-based CTL model checker in VIS-2.3 [18] (with and without preliminary reachability analysis) and, for the properties that can be expressed in LTL, to the FAIR and IC3 algorithms [4, 6]. The total run times were: 27613 s for IICTL; 32220 and 31555 s for VIS with and without reachability analysis.

Table 2 shows for each of the three CTL algorithms the numbers of timeouts (TO) and the numbers of properties that could be solved by only one technique (US). Only the models for which timeouts occurred are listed. While IICTL

Table 2. Timeouts and Unique Solves (nr = no reachability)

model	IICCTL		VIS		VIS-nr		model	IICCTL		VIS		VIS-nr	
	TO	US	TO	US	TO	US		TO	US	TO	US	TO	US
am2901			1	2		1	rgraph	1					
am2910				3			tarb16	16	16			16	
CAB	9				9	11	soap	10		10		10	
checkers	46	6	52			52	swap	2					
cube				1			vcordic	1		1		1	
gcd	2						viper	1		3			
newnim	4						vsa16a	2		2		1	1
palu				5			vsaR			2			
redCAB	5				5	8	vsyst	1		1		1	
rether	1				1	1	total	85	23	98	15	102	1

obtains the lowest number of timeouts and the highest number of unique solutions, it is apparent that the three methods have different strengths and thus are complementary. This point is further brought out by the plots of Figure 3.

The upper row of Figure 8 shows the comparison of IICTL to VIS in the form of scatterplots. The lower left plot compares IICTL to FAIR and IC3 for 110 properties expressible in both CTL and LTL. IC3 is used for safety properties, and FAIR is used for the other ones. Finally, the lower right plot compares the best of IICTL and IC3/FAIR to the best result obtained by VIS, with or without reachability. Even with the averaging effect of taking the best results between two methods, there remain significant differences between the incremental, inductive approach and the one based on BDDs.

The data shown for IICTL were obtained with a medium level of effort in trace generalization (option (2) in Section 3.2). This approach proved the most robust and time-effective, though occasionally, the highest level of effort pays off. This is more likely to be the case when precise reachability information is crucial (e.g., with *rether*, which has very few reachable states).

The comparison of IICTL to the automata-based approach that uses IC3 or FAIR as decision procedure shows that IICTL is close in performance to techniques that are specialized for one type of property. There are three properties for which IICTL times out but IC3 does not. (They are all safety properties.) IICTL gets mired in difficult global reachability queries, because the current

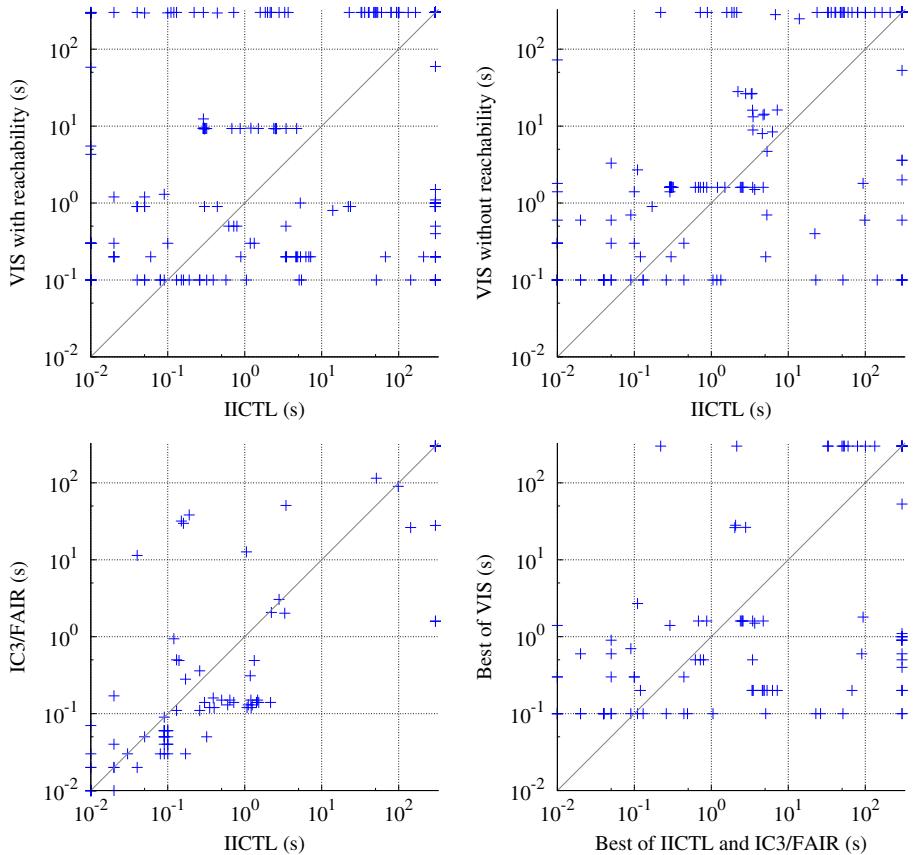


Fig. 3. Comparing IICCTL to competing techniques

implementation does not know that the set of target states will eventually prove empty. On the other hand, since the properties are easily strengthened to inductive, IC3 terminates quickly. These comparisons highlight areas in which progress should be made by making IICCTL’s strategy more flexible and nuanced.

5 Conclusion

Building on the ideas of incremental, inductive verification (IIV) pioneered in IC3 and FAIR, IICCTL is a new property-directed abstracting model checker for CTL with fairness. Although the implementation is preliminary, the experimental results show that it is competitive in robustness and, importantly, complementary to the traditional symbolic BDD-based algorithm. IICCTL offers a different approach to model checking that allows it to prove some properties on systems for which, like `checkers`, BDDs are unwieldy. The techniques for handling CTL properties in an IIV style—the task-based algorithm structured around the parse

graph of the CTL property, and forall-exists generalization of traces—will contribute to the next goal for IIV: CTL* model checking.

References

- [1] Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic Reachability Analysis Based on SAT-Solvers. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 411–425. Springer, Heidelberg (2000)
- [2] Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for CTL*. In: LICS, pp. 388–397 (June 1995)
- [3] Bradley, A.R.: k-step relative inductive generalization. Technical report, CU Boulder (March 2010), <http://arxiv.org/abs/1003.3649>
- [4] 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)
- [5] Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD, pp. 173–180 (November 2007)
- [6] Bradley, A.R., Somenzi, F., Hassan, Z., Zhang, Y.: An incremental approach to model checking progress properties. In: FMCAD, pp. 144–153 (November 2011)
- [7] Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: FMCAD, pp. 135–143 (November 2011)
- [8] 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)
- [9] Du, X., Smolka, S.A., Cleaveland, R.: Local model checking and protocol analysis. STTT 3(1), 219–241 (1999)
- [10] Larsen, K.G.: Proof systems for Hennessy-Milner logic with recursion. TCS 72(2–3), 265–288 (1990)
- [11] <http://iimc.colorado.edu>
- [12] McMillan, K.L.: Symbolic Model Checking. Kluwer, Boston (1994)
- [13] McMillan, K.L.: Applying SAT Methods in Unbounded Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
- [14] Oshman, R.: Bounded model-checking for branching-time logic. Master’s thesis, Technion, Haifa, Israel (June 2008)
- [15] Quielle, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
- [16] Ravi, K., Somenzi, F.: Minimal Assignments for Bounded Model Checking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 31–45. Springer, Heidelberg (2004)
- [17] Stirling, C., Walker, D.: Local model checking in the modal μ -calculus. TCS 89(1), 161–177 (1991)
- [18] <http://vlsi.colorado.edu/~vis>
- [19] Wang, B.-Y.: Proving $\forall\mu$ -Calculus Properties with SAT-Based Model Checking. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 113–127. Springer, Heidelberg (2005)
- [20] Woźna, B.: ACTL* properties and bounded model checking. Fundamenta Informaticae 63(1), 65–87 (2004)

Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement

Matthew Fredrikson¹, Richard Joiner¹, Somesh Jha¹, Thomas Reps^{1,2},
Phillip Porras³, Hassen Saïdi³, and Vinod Yegneswaran³

¹ University of Wisconsin

² Grammatech, Inc.

³ SRI International

Abstract. Stateful security policies—which specify restrictions on behavior in terms of temporal safety properties—are a powerful tool for administrators to control the behavior of untrusted programs. However, the runtime overhead required to enforce them on real programs can be high. This paper describes a technique for *rewriting* programs to incorporate runtime checks so that all executions of the resulting program either satisfy the policy, or halt before violating it. By introducing a rewriting step before runtime enforcement, we are able to perform static analysis to optimize the code introduced to track the policy state. We developed a novel analysis, which builds on abstraction-refinement techniques, to derive a set of runtime policy checks to enforce a given policy—as well as their placement in the code. Furthermore, the abstraction refinement is tunable by the user, so that additional time spent in analysis results in fewer dynamic checks, and therefore more efficient code. We report experimental results on an implementation of the algorithm that supports policy checking for JavaScript programs.

1 Introduction

In this paper, we describe a technique that in-lines enforcement code for a broad class of stateful security policies. Our algorithm takes a program and a policy, represented as an automaton, and re-writes the program by inserting low-level policy checks to ensure that the policy is obeyed. Our key insight is that methods adapted from abstraction-refinement techniques [6] used in software model checking can be applied to optimize the in-lined code. From a security perspective, our approach means that some programs cannot be verified entirely *a priori*, but it allows us to ensure that any program that is executed will always satisfy the policy. Additionally, by bounding the size of the abstraction used in the optimization phase, the tradeoff between static analysis complexity and optimality of the in-lined code can be fine-tuned. The simplicity of this approach is attractive, and allows our algorithm to benefit from advances in the state-of-the-art in automatic program abstraction and model checking.

We implemented our approach for JavaScript, and applied it to several real-world applications. We found the abstraction-refinement approach to be effective at reducing the amount of instrumentation code necessary to enforce stateful

policies. In many cases, all of the instrumentation code can be proven unnecessary after the analysis learns a handful (one or two) facts about the program through counterexamples. In such cases, the analysis has established that the program is safe to run as is, and thus there is no runtime overhead. In cases where the program definitely has a policy violation that the analysis uncovers, instrumentation is introduced to exclude that behavior (by causing the program to halt before the policy is violated), again using only a few facts established by static analysis.

To summarize, our contributions are:

- A language-independent algorithm for weaving stateful policies into programs, to produce new programs whose behavior is identical to the original on all executions that do not violate the policy.
- A novel application of traditional software model-checking techniques that uses runtime instrumentation to ensure policy conformity whenever static analysis is too imprecise or expensive. The degree to which the analysis relies on static and dynamic information is tuneable, which provides a trade-off in runtime policy-enforcement overhead.
- A prototype implementation of our algorithm for JavaScript, called JAM, and an evaluation of the approach on real JavaScript applications. The evaluation validates our hypothesis that additional time spent in static analysis, utilizing the abstraction-refinement capabilities of our algorithm, results in fewer runtime checks. For five of our twelve benchmark applications, learning just four predicates allows JAM to place an optimal number of runtime checks necessary to enforce the policy.

The rest of the paper is laid out as follows. Section 2 gives an overview of the algorithm. Section 3 presents the technical details of the analysis and discuss JAM. Section 4 evaluates the performance of JAM over a set of real applications. Section 5 discusses related work.

2 Overview

We propose a hybrid approach to enforcing policies over code from an untrusted source. Our solution is to perform as much of the enforcement as possible statically, and to use runtime checks whenever static analysis becomes too expensive. This approach allows us to avoid overapproximations on code regions that are difficult to analyze statically. Furthermore, varying the degree to which the analysis relies on runtime information allows us to control the cost of static analysis at the expense of performing additional runtime checks. While this approach means that many programs cannot be verified against a policy *a priori* before execution, an interpreter provided with the residual information from the static analysis can prevent execution of any code that violates the policy. In fact, as we show in Section 3, the target program can often be rewritten to in-line any residual checks produced by the static analysis, sidestepping the need for explicit support from the interpreter.

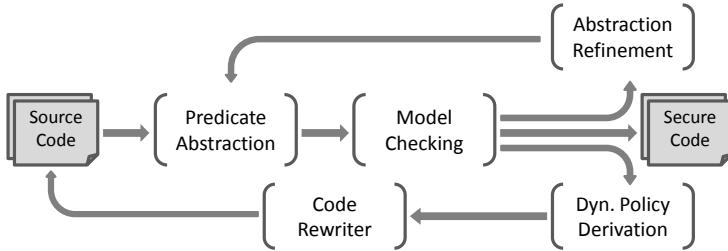


Fig. 1. Workflow overview of our approach

Figure 1 presents an overview of our approach. The security policy is first encoded as a temporal safety property over the states of the target program. The algorithm then begins like other software model checkers by first performing predicate abstraction [12] over the target code, and checking the resulting model for a reachable policy-violating state [20]. Our algorithm differs from previous techniques in how the results of the model checker are used; when the model checker produces an error trace, there are a few possibilities.

1. If the trace is valid, then our algorithm places a dynamic check in the target code to prevent it from being executed on any concrete path.
2. If the trace is not valid, then the algorithm can either:
 - (a) Refine the abstraction and continue model checking.
 - (b) Construct a dynamic check that blocks execution of the trace *only* when the concrete state indicates that it will violate the policy.

Item (1) has the effect of subtracting a known violating trace from the behaviors of the program, and in general, some number of *similar* behaviors, thereby decreasing the size of the remaining search space. For an individual counterexample, item (2)(a) follows the same approach used in traditional counterexample-guided abstraction refinement-based (CEGAR) software model checking. Item (2)(b) ensures that a potentially-violating trace identified statically is never executed, while avoiding the expense of constructing a proof for the trace. However, the inserted check results in a runtime performance penalty—thus, the choice corresponds to a configurable tradeoff in analysis complexity versus runtime overhead.

To illustrate our approach, consider the program listed in Figure 2(a). This code is a simplified version of the sort of dispatching mechanism that might exist in a command-and-control server [24] or library, and is inspired by common JavaScript coding patterns. The function `execute` takes an instruction code and data argument, and invokes an underlying system API according to a dispatch table created by the program’s initialization code.

We will demonstrate enforcement of the policy given in Figure 2(c), which is meant to prevent exfiltration of file and browser-history data. Observe that we specify this policy, which corresponds to a temporal safety property, using an automaton that summarizes all of the “bad” paths of a program that might lead to a violation. Thus, policies encode properties on individual paths of a program,

```

1 api[0] = readFile;
2 api[1] = sendPacket;
3 fun execute(instr, data) {
4   api[instr](data);
5 }
6 while(*) {
7   instr, data = read();
8   execute(instr, data);
9 }

```

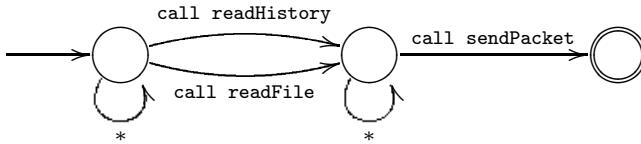
(a) Original code (P).

```

1 policy = 0;
2 api[0] = readFile;
3 api[1] = sendPacket;
4 fun execute(instr, data) {
5   if(api[instr] == readFile
6     && policy == 0) policy++;
7   if(api[instr] == sendPacket
8     && policy == 1) halt();
9   api[instr](data);
10 }

```

(b) Safe code (previously safe parts omitted). Shaded lines contain two checks inserted by our algorithm; our analysis prevented an additional check before line 9.



(c) Security policy that says “do not read from a file or the history and subsequently write to the network.”

Fig. 2. Dangerous code example

```

api[0] = readFile;
api[1] = sendPacket;
instr, data = read();
execute(0, data);
assume{api[0] == readFile}
api[instr](data);
instr, data = read();
execute(1, data);
assume{api[1] == sendPacket}
api[instr](data);

```

(1)

```

api[0] = readFile;
api[1] = sendPacket;
instr, data = read();
execute(0, data);
assume{api[0] == readHistory}
api[instr](data);
instr, data = read();
execute(1, data);
assume{api[1] == sendPacket}
api[instr](data);

```

(2)

Fig. 3. Counterexamples returned from the example in Figure 2 (2) is invalid, and leads to a spurious runtime check.

and we intuitively think of policy violation as occurring when a subsequence of statements in a path corresponds to a word in the language of the policy automaton. The example in Figure 2(c) is representative of the policies used by our analysis, and meant to convey the important high-level concepts needed to understand our approach. For more details about specific policies, see Section 3.

To verify this policy against the program, we proceed initially with software model checking. The first step is to create an abstract model of the program using a finite number of relevant predicates [12]. We begin with three predicates, corresponding to the states relevant to the policy: `@Func = readHistory`, `@Func = readFile`, and `@Func = sendPacket`. We assume the existence of a special state variable `@Func`, which holds the current function being executed; each of these predicates queries which function is currently being executed. With

these predicates, the software model checker will return the two counterexamples shown in Figure 3.

(1) is valid — it corresponds to a real policy violation. Our analysis updates the program’s code by simulating the violating path of the policy automaton over the actual state of the program. This is demonstrated in Figure 2(b); the automaton is initialized on line 1, and makes a transition on lines 6 and 8 after the program’s state is checked to match the labels on the corresponding policy transitions. When the transition to the final state is taken, the program halts.

(2) is not valid—the assumption that `api[0] == readHistory` never holds. However, for the analysis to prove this, it would need to learn a predicate that encodes this fact, and build a new, refined abstraction on which it can perform a new statespace search. If the user deems this too expensive, then the analysis can simply insert another runtime check before line 7 corresponding to the first transition in the policy automaton, which increments `policy` whenever `api[instr]` holds `readHistory`. This approach will result in a secure program, but will impose an unnecessary runtime cost: every time `execute` is called, this check will be executed but the test will never succeed. Alternatively, the analysis can learn the predicates `{api[instr] = readHistory, instr = 0}`, and proceed with software model checking as described above. This will result in the more efficient code shown in Figure 2(b).

3 Technical Description

Our analysis takes a program \mathcal{P} and a policy, and produces a new program \mathcal{P}' by inserting policy checks at certain locations in \mathcal{P} needed to ensure that the policy is not violated. In this section, we describe the policies that we support, as well as the algorithm for inserting policy checks. The algorithm we present has several important properties that make it suitable for practical use:

1. Upon completion, it has inserted a complete set of runtime checks necessary to enforce the policy: any program that would originally violate the policy is guaranteed not to violate the policy after rewriting. Runs that violate the policy must encounter a check, and are halted before the violation occurs.
2. The policy checks inserted will not halt execution unless the current execution of the program will inevitably lead to a policy violation. In other words, *our approach does not produce any false positives*.
3. The running time of the main algorithm is bounded in the size of the program abstraction, which is controllable by the user. This approach yields a trade-off between program running time and static-analysis complexity.
4. JAM always terminates in a finite number of iterations.

We begin with a description of our problem, and proceed to describe our language-independent algorithm for solving it (Section 3.1). The algorithm relies only on standard semantic operators, such as symbolic precondition and abstract statespace search. In Section 3.2, we discuss our implementation of the algorithm for JavaScript.

3.1 Runtime Checks for Safety Properties

Preliminaries. A run of a program \mathcal{P} executes a sequence of statements, where each statement transforms an element $\sigma \in \Sigma_{\mathcal{P}}$ (or *state*) of \mathcal{P} 's state space to a new, not necessarily distinct, state σ' . For the purposes of this section, we do not make any assumptions about the form of \mathcal{P} 's statements or states. We use a labeling function ι for each statement s in \mathcal{P} , so that $\iota(s)$ denotes a unique integer. Let a *state trace* be a sequence of states allowed by \mathcal{P} , and let $\mathcal{T}_{\mathcal{P}} \subseteq \Sigma^*$ be the complete set of possible state traces of \mathcal{P} .

The policies used by our analysis are based on *temporal safety properties*. A temporal safety property encodes a finite set of sequences of events that are not allowed in any execution of the program. We represent these properties using automata.¹ The events that appear in our policies correspond to concrete program states, and we do not allow non-trivial cycles among the transitions in the automaton.

Definition 1 (*Temporal safety automaton*). A *temporal safety automaton* Φ is an automaton $\Phi = (Q, Q_s, \delta, Q_f, \mathcal{L})$ where

- Q is a set of states (with $Q_s \subseteq Q$ and $Q_f \subseteq Q$ the initial and final states, respectively). Intuitively, each $q \in Q$ represents sets of events that have occurred up to a certain point in the execution.
- $\delta \subseteq Q \times \mathcal{L} \times Q$ is a deterministic transition relation that does not contain any cycles except for self-loops.
- \mathcal{L} is a logic whose sentences represent sets of program states, i.e., $\phi \in \mathcal{L}$ denotes a set $[\![\phi]\!] \subseteq \Sigma$.

For a given $(q, \phi, q') \in \delta$, the interpretation is that execution of a statement from a program state σ where ϕ holds (i.e., $\sigma \in [\![\phi]\!]$) causes the temporal safety automaton to transition from q to q' . Self-loops are necessary to cover statements that do not affect the policy state, but other types of cycles can prevent our algorithm from terminating in a finite number of iterations. This leads to the definition of property matching: a program \mathcal{P} **matches** a temporal safety automaton Φ , written $\mathcal{P} \models \phi$, if it can generate a state trace that matches a word in the language of the automaton.

Our problem is based on the notion of property matching. Given a program \mathcal{P} and temporal safety automaton Φ , we want to derive a new program \mathcal{P}' that: (i) does not match Φ , (ii) does not contain any *new* state traces, and (iii) preserves all of the state traces from \mathcal{P} that do not match Φ .

Policy Checks from Proofs. Our algorithm is shown in Algorithm 11, and corresponds to the workflow in Figure 11. SafetyWeave takes a program \mathcal{P} , a finite set of predicates E from \mathcal{L} , a bound on the total number of predicates k , and a temporal safety automaton policy Φ . We begin by using predicate set E to build a sound abstraction of \mathcal{P} [12] (this functionality is encapsulated by `Abs` in Algorithm 11). Note that E must contain a certain set of predicates, namely

¹ This formalism is equivalent to past-time LTL.

Algorithm 1. SafetyWeave(\mathcal{P}, E, k, Φ)

Require: $k \geq 0$
Require: $\phi \in E$ for all $(q, \phi, q') \in \delta$

```

repeat
     $\mathcal{P}_E \leftarrow \text{Abs}(\mathcal{P}, E)$  {Build abstraction}
     $\pi \leftarrow \text{IsReachable}(\mathcal{P}_E, \Phi)$ 
    if  $\pi = \text{NoPath}$  then
        return  $\mathcal{P}$ 
    else if  $\text{isValid}(\pi)$  then
        {Build runtime policy check; rewrite P to enforce it}
         $\Psi_D^\pi \leftarrow \text{BuildPolicy}(\mathcal{P}, \pi, \Phi)$ 
         $\mathcal{P} \leftarrow \text{Enforce}(\mathcal{P}, \Psi_D^\pi)$ 
    else
        {Refine the abstraction}
        if  $|E| < k$  then
             $E \leftarrow E \cup \text{NewPreds}(\pi)$ 
        else
            {We have reached saturation of the abstraction set E}
            {Build runtime policy check; rewrite P to enforce it}
             $\Psi_D^\pi \leftarrow \text{BuildPolicy}(\mathcal{P}, \pi, \Phi)$ 
             $\mathcal{P} \leftarrow \text{Enforce}(\mathcal{P}, \Psi_D^\pi)$ 
        end if
    end if
until forever

```

ϕ_i for each $(q_i, \phi_i, q'_i) \in \delta^\Phi$. The abstraction is then searched for traces from initial states to bad states (encapsulated by `IsReachable`), which correspond to final states in Φ . If such a trace π is found, it is first checked to see whether it corresponds to an actual path through \mathcal{P} (performed by `IsValid`). If it does, or if we cannot build an abstraction that does not contain π , then a runtime policy check Ψ_D^π is derived (encapsulated by `BuildPolicy`) and added to \mathcal{P} (performed by `Enforce`). Ψ_D^π identifies a concrete instance of π .

If π does not correspond to an actual policy-violating path of \mathcal{P} , and we have fewer than k predicates, then the abstraction is refined by learning new predicates (encapsulated by `NewPreds`). Otherwise, we add a runtime check to prevent the concrete execution of π . This process continues until we have either proved the absence of violating paths (via abstraction refinement), or added a sufficient set of runtime checks to prevent the execution of possible violating paths.

Termination. SafetyWeave is guaranteed to terminate in a finite number of iterations, due to the following properties: (i) the algorithm will stop trying to prove or disprove the validity of a single trace after a finite number of iterations, due to the bounded abstraction size ($|E|$ is limited by k). (ii) In the worst case, it must insert a policy check for each transition in Φ before every statement in \mathcal{P} . Once \mathcal{P} is thus modified, `IsReachable` will not be able to find a violating trace π , and will terminate.

Abstracting \mathcal{P} (Abs). On each iteration, an abstraction \mathcal{P}_E is built from the predicate set E and the structure of \mathcal{P} . \mathcal{P}_E has two components: a control automaton G_C and a data automaton G_D . Each automaton is a nested word automaton (NWA) [2] whose alphabet corresponds to the set of statements used in \mathcal{P} . G_C overapproximates the set of all paths through \mathcal{P} that are valid with respect to \mathcal{P} 's control structure (i.e., call/return nesting, looping, etc.), whereas G_D overapproximates the paths that are valid with respect to the data semantics of \mathcal{P} . In G_C , states correspond to program locations, and each program location corresponds to the site of a potential policy violation, so each state is accepting. In the data automaton, states correspond to sets of program states, and transitions are added according to the following rule: given two data-automaton states q and q' representing ϕ and ϕ' , respectively, the G_D contains a transition from q to q' on statement s whenever $\phi \wedge \text{Pre}(s, \phi')$ is satisfiable, where Pre is the symbolic precondition operator. We then have that $L(G_C) \cap L(G_D)$ represents an overapproximation of the set of valid paths through \mathcal{P} ; this is returned by Abs.

Separating the abstraction into G_C and G_D allows us to provide a straightforward, well-defined interface for extending the algorithm to new languages. To be able to instantiate Algorithm II to work on programs written in a different language, a tool designer need only provide (i) a symbolic pre-image operator for that language to build G_D , and (ii) a generator of interprocedural control-flow graphs (ICFGs) to build G_C .

Proposition 1 $L(G_D)$ corresponds to a superset of the traces of \mathcal{P} that might match Φ .

Checking the Abstraction (IsReachable). Given an automaton-based abstraction $\mathcal{P}_E = G_C \cap G_D$, IsReachable finds a path in \mathcal{P}_E that matches Φ . This operation is encapsulated in the operator \cap_{pol} specified in Definition 2. In essence, Definition 2 creates the product of two automata— \mathcal{P}_E and Φ . However, the product is slightly non-standard because \mathcal{P}_E has an alphabet of program statements, whereas Φ has an alphabet of state predicates. Note that when we refer to the states of \mathcal{P}_E in Definition 2, we abuse notation slightly by only using the program state component from G_D , and dropping the program location component from G_C .

Definition 2 (Policy Automaton Intersection \cap_{pol}). Given a temporal safety automaton $\Phi = (Q^\Phi, Q_s^\Phi, \delta^\Phi, Q_f^\Phi)$ and an NWA $G = (Q^G, Q_s^G, \delta^G, Q_f^G)$ whose states correspond to sets of program states, $G \cap_{\text{pol}} \Phi$ is the nested word automaton (Q, Q_s, δ, Q_f) , where

- Q has one element for each element of $Q^G \times Q^\Phi$.
- $Q_s = \{(\phi, q^\Phi) \mid q^\Phi \in Q_s^\Phi, \phi \in Q_s^G\}$, i.e., an initial state is initial in both G_D and Φ .
- $\delta = \langle \delta_{\text{in}}, \delta_{\text{ca}}, \delta_{\text{re}} \rangle$ are the transition relations with alphabet \mathcal{S} . For all $(q^\Phi, \phi'', q'^\Phi) \in \delta^\Phi$, and $\phi, \phi' \in Q^G$ such that $\phi \wedge \text{Pre}(s, \phi' \wedge \phi'')$ is satisfiable, we define each transition relation using the transitions in $\delta^G = (\delta_{\text{in}}^G, \delta_{\text{ca}}^G, \delta_{\text{re}}^G)$:
 - δ_{in} : when $(\phi, s, \phi') \in \delta_{\text{in}}^G$, we update δ_{in} with: $((\phi, q^\Phi), s, (\phi', q'^\Phi))$.

- δ_{ca} : when $(\phi, s, \phi') \in \delta_{\text{ca}}^G$, we update δ_{ca} with: $((\phi, q^\Phi), s, (\phi', q'^\Phi))$.
- δ_{re} : when $(\phi, \phi''', s, \phi') \in \delta_{\text{re}}^G$, we update δ_{re} with: $((\phi, q^\Phi), (q'''^\Phi, \phi'''), s, (\phi', q'^\Phi))$ for all $q'''^\Phi \in Q_f^\Phi$.
- $Q_f = \{(\phi, q^\Phi) \mid q^\Phi \in Q_f^\Phi, \phi \in Q_f^G\}$, i.e., a final state is final in Φ and G .

The words in $L(\mathcal{P}_E \cap_{\text{pol}} \Phi)$ are the sequences of statements (traces) in \mathcal{P} that respect the sequencing and nesting specified in the program, and may lead to an error state specified by Φ . As long as G_C and G_D overapproximate the valid traces in \mathcal{P} , we are assured that if an erroneous trace exists, then it will be in $L(\mathcal{P}_E \cap_{\text{pol}} \Phi)$. Additionally, if $L(\mathcal{P}_E \cap_{\text{pol}} \Phi) = \emptyset$, then we can conclude that \mathcal{P} cannot reach an error state.

Checking Path Validity (IsValid); Refining the Abstraction (NewPreds). Given a trace $\pi \in L(\mathcal{P}_E)$, we wish to determine whether π corresponds to a possible path through \mathcal{P} (i.e., whether it is *valid*). This problem is common to all CEGAR-based software model checkers [316], and typically involves producing a formula that is valid iff π corresponds to a real path. We discuss an implementation of `IsValid` for JavaScript in Section 3.2.

Because \mathcal{P}_E overapproximates the error traces in \mathcal{P} , two conditions can hold for a trace π . (i) The sequence of statements in π corresponds to a valid path through \mathcal{P} that leads to a violation according to Φ , or it cannot be determined whether π is a valid trace or not. (ii) The sequence of statements in π can be proven to be invalid. In the case (i), a runtime check is added to \mathcal{P} to ensure that the error state is not entered at runtime (see the following section for a discussion of this scenario). In the case of (ii), \mathcal{P}_E is refined by adding predicates to G_D (encapsulated in the call to `NewPreds`). Standard techniques from software model checking may be applied to implement `NewPreds`, such as interpolation [25] and unsatisfiable-core computation [16]; we discuss our JavaScript-specific implementation in Section 3.2.

Deriving and Enforcing Dynamic Checks. The mechanism for deriving dynamic checks that remove policy-violating behavior is based on the notion of a *policy-violating witness*. A policy-violating witness is computed for each counterexample trace produced by the model checker that is either known to be valid, or cannot be validated using at most k predicates. A policy-violating witness must identify at runtime the concrete instance of the trace π produced by the model checker before it violates the policy Φ . To accomplish this, we define a policy-violating witness as a sequence containing elements that relate statements to assertions from Φ . The fact that a check is a sequence, as opposed to a set, is used in the definition of `Enforce`.

Definition 3 (Policy-violating witness). A *policy-violating witness* $\Psi_\Phi^\pi \in (\mathbb{N} \times \mathcal{L})^*$ for a trace π and policy Φ is a sequence of pairs relating statement elements in π to formulas in \mathcal{L} . We say that $\pi' \models \Psi_\Phi^\pi$ (or π' matches Ψ_Φ^π) if there exists a subsequence π'' of π' that meets the following conditions:

1. The statements in π'' match those in Ψ_Φ^π : $|\pi''| = |\Psi_\Phi^\pi|$, and for all $(i, \phi_i) \in \Psi_\Phi^\pi$, $\iota^{-1}(i)$ is in π'' .

2. Immediately before \mathcal{P} executes a statement s corresponding to the i^{th} entry of Ψ_Φ^π (i.e. $(\iota(s), \phi_i)$), the program state satisfies ϕ_i .

Suppose that $\Phi = (Q^\Phi, Q_i^\Phi, \delta^\Phi, Q_f^\Phi)$ is a temporal safety automaton, and π is a path that causes \mathcal{P} to match Φ . Deriving Ψ_Φ^π proceeds as follows: because π is a word in $L(\mathcal{P}_E = G_D \cap G_C)$, there must exist some subsequence $s_{i_1} s_{i_2} \dots s_{i_m}$ of π that caused transitions between states in Φ that represent distinct states in Φ . We use those statements, as well as the transition symbols $[\phi_i]_{i \in \{i_1, i_2, \dots, i_m\}}$ from Φ on the path induced by those statements, to build the j^{th} element of Ψ_Φ^π by forming pairs (i_j, ϕ_j) , where the first component ranges over the indices of $s_{i_1} s_{i_2} \dots s_{i_m}$.

More precisely, for all $i \in i_1, i_2, \dots, i_m$, there must exist $(q_i, \phi_i, q'_i) \in \delta^\Phi$ and $((\phi, q_i), s, (\phi', q'_i)) \in \delta^{\mathcal{P}_E \cap \text{pol}^\Phi}$ such that $\phi' \wedge \phi_i$ is satisfiable (recall the \cap_{pol} from Definition 2). Then:

$$\Psi_\Phi^\pi = [(i_{i_1}, \phi_1), (i_{i_2}, \phi_2), \dots, (i_{i_m}, \phi_m)]$$

Intuitively, Ψ_Φ^π captures the statements in π responsible for causing Φ to take transitions to its accepting state, and collects the associated state assertions to form the policy-violating witness.

We now turn to **Enforce**, which takes a policy-violating witness Ψ_Φ^π , and a program \mathcal{P} , and returns a new program \mathcal{P}' such that \mathcal{P}' does not contain a path π such that $\pi \models \Psi_\Phi^\pi$. The functionality of **Enforce** is straightforward: for each element (i, ϕ) in Ψ_Φ^π , insert a guarded transition immediately before $\iota^{-1}(i)$ to ensure that ϕ is never true after executing $\iota^{-1}(i)$. The predicate on the guarded transition is just the negation of the precondition of ϕ with respect to the statement $\iota^{-1}(i)$, and a check that the *policy variable* (inserted by **Enforce**) matches the index of (i, ϕ) in Φ . When the guards are true, the statement either increments the policy variable, or halts if the policy variable indicates that all conditions in Ψ_Φ^π have passed. A concrete example of this instrumentation in Figure 2.

Note that a given occurrence of statement s in \mathcal{P} may be visited multiple times during a run of \mathcal{P} . Some subset of those visits may cause Φ to transition to a new state. In this scenario, notice that our definition of **Enforce** will insert multiple guarded transitions before s , each differing on the condition that they check—namely, each transition (q, ϕ, q') of Φ that was activated by s in the policy-violating witness will have a distinct condition for $\text{Pre}(s, \phi)$ that either increments the policy variable or halts the program. Additionally, the check on the policy variable in each guard prevents the policy variable from being updated more than once by a single check.

Definition 4 (Functionality of Enforce). Given a program \mathcal{P} and a dynamic check $\Psi_\Phi^\pi = \{(i_1, \phi_1), \dots, (i_n, \phi_n)\}$, **Enforce** produces a new program \mathcal{P}' . \mathcal{P}' uses a numeric variable, *policy*, which is initialized to zero. **Enforce** performs the following steps for each element $(i, \phi) \in \Psi_\Phi^\pi$:

1. Let $\phi_{\text{pre}} \equiv \text{Pre}(\iota^{-1}(i), \phi) \wedge \text{policy} = j$, where j is the index of (i, ϕ) in Φ_Φ^π .
2. Insert a new statement before $\iota^{-1}(i)$ that either:

- Increments **policy** whenever ϕ_{pre} is true and $\text{policy} < |\Psi_\Phi^\pi|$.
- Halts the execution of \mathcal{P}' whenever ϕ_{pre} is true and $\text{policy} = |\Psi_\Phi^\pi|$.

For **Enforce** to operate correctly, \mathcal{L} must be closed under the computation of pre-images, and pre-images of formulas in \mathcal{L} must be convertible to code in the target language. When **Enforce** is called on all counterexample paths returned by Algorithm 1, the resulting program will not match Φ .

3.2 JavaScript Prototype

We implemented our algorithm for JavaScript, in a tool called JAM. There are four components to Algorithm 1 that must be made specific to JavaScript: the control (G_C) and data (G_D) automaton generators (**Abs**), the path validity checker (**IsValid**), and the predicate learner (**NewPreds**). To build the control automaton, we used Google’s Closure Compiler [17], which contains methods for constructing an intraprocedural control flow graph (CFG) for each function in a program, as well as dataflow analyses for determining some of the targets of indirect calls. The only language-specific aspect of the data-automaton generator is the computation of symbolic pre-state for a given statement in \mathcal{P} . We use Maffei et al.’s JavaScript operational semantics [21], lifted to handle symbolic term values, and implemented as a set of Prolog rules. Computing a satisfiability check to build G_D in this setting amounts to performing a query over this Prolog program, with ground state initialized to reflect the initial state of the program. To learn new predicates (i.e., to compute **NewPreds**), we apply a set of heuristics to the failed counterexample trace that we have developed from our experience of model checking real JavaScript programs. Our heuristics are based on the statement that made the trace invalid; the predicate they build depends on the type of that statement (e.g., if the statement is an **if** statement, the new predicate will be equivalent to the statement’s guard expression).

Currently, the JAM implementation does not handle programs that contain dynamically generated code—e.g., generated via language constructs, such as `eval()` or `Function()`, or via DOM interfaces, such as `document.write()`. JAM currently only handles a subset of the DOM API that most browsers support. None of these are fundamental limitations, although supporting dynamically generated code soundly could cause a large number of runtime checks to be introduced. Dynamically generated code can be supported by inserting code that updates the state of the **policy** variable (Definition 1) by simulating the policy automaton before each dynamically generated statement, in the manner of Erlingsson et al. [8]. Additional DOM API functions can be supported by adding reduction rules to our semantics that capture the behavior of the needed DOM API.

4 Experimental Evaluation

In this section, we summarize the performance and effectiveness of JAM in applying realistic security policies to ten JavaScript applications (plus alternative

versions of two of them that we seeded with policy-violating code). The results, summarized in Table II, demonstrate that the time devoted to static analysis during the abstraction-refinement stage often leads to fewer runtime checks inserted into the subject program. Additionally, because the CEGAR process evaluates the validity of the potentially-violating execution traces found in the abstract model, time spent during this stage also yields greater confidence that the inserted checks are legitimately needed to prevent policy violations during runtime.

The benchmark applications used to measure JAM’s performance are real programs obtained from the World Wide Web. Consequently, the policies we developed typically address cross-domain information-leakage issues and data-privacy issues that are of concern in that domain. Our measurements indicate that under such realistic circumstances, (i) JAM is able to identify true vulnerabilities while (ii) reducing spurious dynamic checks, and (iii) is able to do so with analysis times that are not prohibitive.

4.1 Results

Because the goal of the system is to derive a minimal set of runtime checks needed to ensure adherence to a policy, we sought to measure the benefits of refining the program model against the cost of performing such analysis. This information was gathered by comparing the running time and result of JAM’s analysis under varying levels of abstraction refinement, achieved by placing a limit on the number of predicates learned during the CEGAR analysis before proceeding to the saturation phase. The validation of counterexamples and learning of new predicates can be disabled altogether, which establishes the baseline effectiveness of static analysis without abstraction refinement. Measurements of JAM’s effectiveness and efficiency with different levels of abstraction refinement are presented in Table I.

One dimension on which to evaluate the behavior of JAM is the number of necessary versus spurious checks that it inserts. All checks that are inserted during the CEGAR phase are known to be necessary, because the abstract counterexample that gave rise to each such check has been proven valid. In contrast, spurious checks may be inserted in the saturation phase. We inspected the applications manually to determine the number of necessary checks Columns 5 and 6 of Table 2 classify the checks identified during saturation as valid or spurious according to our manual classification. A lower number of spurious checks inserted under a particular configuration represents a more desirable outcome vis a vis minimizing runtime overhead.

Reported performance statistics are the averages of multiple runs on a VirtualBox VM running Ubuntu 10.04 with a single 32-bit virtual processor and 4GB memory. The host system is an 8-core HP Z600 workstation with 6GB memory running Red Hat Enterprise Linux Server release 5.7. Execution time and memory usage refer to the total CPU time and maximum resident set size as reported by the GNU time utility version 1.7.

The results for `flickr` demonstrate the benefit of additional effort spent on abstraction refinement. Analysis of the unrefined model identifies two potential violations of the policy, one of which is spurious and the other valid (according

Table 1. Performance of JAM on selected benchmarks. *Learned* denotes the number of predicates learned through abstraction refinement, *Total* to the number of learned predicates plus those in the initial state from the policy. *CEGAR* denotes the number of checks placed before the abstraction size limit is reached, *Saturation* to those placed afterwards.

Benchmark application	Predicates		Checks			Execution time (s)	Memory (KB)		
	Learned	Total	Saturation		Total				
			CEGAR	Valid	Spurious				
flickr	2	3	1	0	0	1	138.67		
flickr	1	2	1	0	1	2	74.49		
flickr	0	1	0	1	1	2	24.23		
beacon	0	3	2	0	0	2	74.50		
jssec	1	2	0	0	0	0	14.04		
jssec	0	1	0	0	1	1	7.59		

to our manual classification of checks). When allowed to learn a single predicate, JAM is able to avoid a spurious trace, and identify the valid counterexample. Allowing JAM to learn two predicates causes it to prove the spurious counterexample invalid, and rule out the un-needed runtime check.

The policy for the `beacon` benchmark is more involved—using multiple transition sequences to characterize the policy violation; it states “a cookie should never be written after the DOM is inspected using `document.getElementById` or `document.getElementsByTagName`.“ This policy represents a cross-domain information-leakage concern that JAM is able to identify and validate in the first iteration of the analysis. The `jssec` application is intended to allow a website user to open and close a section of the page being viewed. The policy for `jssec` states that the only allowable change to a DOM element’s style properties is to the `display` attribute; otherwise, the code could change the `backgroundImage` attribute, thereby initiating an HTTP call to a remote server. JAM successfully proves that the program is free of violations by learning the prototype of an object whose member is the target of an assignment.

5 Related Work

In-Lined Reference Monitors. In-lined reference monitors were first discussed by Erlingsson and Schneider [8,28] who applied the idea to both Java and x86 bytecode. Their prototype, SASI, supports security policies as finite-state machines with transitions denoting sets of instructions (i.e., predicates over instructions) that may be executed by the untrusted program. Note the distinction from the policy automata used in our work, where transitions have predicates that refer to the program state, *not* just restrictions on the next instruction to execute. SASI works by inserting policy-automaton-simulation code before every instruction in the program, and then uses local simplification to remove as much of the added code as possible. This amounts to applying the available local static information at each location to evaluate the instruction predicate to the greatest degree possible; the authors opted not to use global static analysis in the interest

of maintaining a small TCB. In this respect, the primary focus of our work is quite different from Erlingsson and Schneider’s foundational work.

Since Erlingsson and Schneider’s work, this has been an active area of research. Nachio [9] is an in-lined-monitor compiler for C, where policies are given as state machines with fragments of imperative code that execute at each state. The Java-MOP (Monitor-Oriented Programming) system [5] allows users to choose from a set of temporal logics, domain-specific logics, and languages in which to express policies. ConSpec [1] performs in-lined reference monitoring based on policies similar to those used by Erlingsson and Schneider, and takes the additional step of formally verifying the in-lined monitor. SPoX [14] built on aspect-oriented programming to implement in-lined reference monitoring for Java, using as policies automata whose edges are labeled with pointcut expressions. They define a formal semantics for their policies, laying the groundwork for future work on verified implementations of in-lined reference monitors; this feature can also aid in developing analyses for optimizing the in-lined monitor code, although the authors do not pursue this idea. Sridhar and Hamlen [29] designed an IRM-compiler for JavaScript bytecodes, and showed how software model checking can be applied to verify the compiled in-lined monitor code. Hamlen et al. [15] designed MOBILE, an extension to the .NET runtime that supports IRMs with the advantage that well-typed MOBILE code is guaranteed to satisfy the policy it purports to enforce. The primary difference between these previous efforts and our own is our focus on optimizing in-lined monitor code, and our use of abstraction-refinement techniques to do this in a tuneable manner.

Clara [4] is a framework for incorporating static analysis into the reference-monitor in-lining process. The setting in which Clara operates is similar to ours: an untrusted program and a security policy, represented by a finite-state machine, are provided, and the goal is to produce a rewritten program that always obeys the policy. It works on top of an aspect-weaving framework for Java [18] by first weaving the policy (represented as an aspect) into the program, and subsequently applying a modular set of static analyses to remove as many join points as possible. In this regard, Clara is conceptually similar to our work; it is conceivable that parts of our work could be combined as a path-sensitive, semantics-driven static-analysis component inside of Clara’s modular framework. Otherwise, our work differs from Clara in one important respect: the policies we use provide direct means to refer to the dynamic state of the program, allowing richer and more concise policies. Clara’s dependence on AspectJ limits the building blocks of expressible policies to a pre-defined set of pointcuts.

JavaScript Policy Enforcement. Several recent projects attempt to identify subsets of JavaScript that are amenable to static analysis. Two early examples are ADSafe [7] and FBJS [10], which facilitate “mashups” by removing language elements that make it difficult to isolate the effects of distinct JavaScript programs executing from the same domain. Maffeis *et al.* explored a similar approach [22,23], but took the additional step of formally verifying their subsets against small-step operational semantics of the ECMAScript specification. More recently, Google has released Caja [11], uses the object-capability

model to provide isolation. Our work differs from efforts to identify secure JavaScript subsets for isolation primarily in the class of policies we are able to support. Rather than sandbox-based object-capability policies, JAM can verify arbitrary safety properties, including flow-sensitive temporal-safety properties.

Guarnieri and Livshits presented GATEKEEPER, a “mostly static” JavaScript analysis based on points-to information that is calculated using Datalog inference rules [13]. Unlike JAM, Gatekeeper is not capable of checking flow-insensitive policies, and it is not clear how it can be made flow-sensitive without greatly increasing cost. Kudzu [27] is a JavaScript bug-finding system that uses forward-symbolic execution. This functionality stands in contrast to JAM, as dangerous program paths are reported to the user at analysis time, whereas in JAM they are rewritten to halt at runtime before the dangerous (policy-violating) payload is executed: JAM always inserts sufficient instrumentation to soundly and completely enforce a given policy.

Yu *et al.* proposed a safe browsing framework based on syntax-directed rewriting of the JavaScript source according to an edit automaton [30]. Their work is formalized in terms of a JavaScript subset they call *CoreScript*, which excludes the same difficult language elements as most other static JavaScript analyses. While our current implementation does not support the full language either, this is not a limitation of our approach. The dynamic component of our policy-enforcement method is capable of monitoring the execution of these language elements. The syntax-directed nature of their rewriting framework effectively restricts the class of policies it can enforce. More recently, Meyerovich and Livshits presented ConScript [26], which is an in-browser mechanism for enforcing fine-grained security policies for JavaScript applications. One of the primary contributions of ConScript is a type system for checking policy-instrumentation code against several different types of attack *on the integrity of the policy*. Essentially, ConScript is a system for specifying and implementing advice [19] on JavaScript method invocations. Thus, ConScript is complementary in function to JAM: while JAM takes a high-level logical formula that represents a security policy, and finds a set of program locations to place policy instrumentation, ConScript is capable of soundly and efficiently enforcing that instrumentation on the client side, during execution.

References

1. Aktug, I., Naliuka, K.: Conspec – a formal language for policy specification. ENTCS 197 (February 2008)
2. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56(3) (2009)
3. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL (2002)
4. Bodden, E., Lam, P., Hendren, L.: Clara: A Framework for Partially Evaluating Finite-State Runtime Monitors Ahead of Time. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 183–197. Springer, Heidelberg (2010)
5. Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005)

6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *JACM* 50(5) (2003)
7. Crockford, D.: Adsafe: Making JavaScript safe for advertising, <http://www.adsafe.org>
8. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: *NSPW* (2000)
9. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: *SP* (1999)
10. Facebook, Inc. FBJS, <http://wiki.developers.facebook.com/index.php/FBJS>
11. Google inc. The Caja project, <http://code.google.com/p/google-caja/>
12. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) *CAV* 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
13. Guarneri, S., Livshits, B.: Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In: *Security* (August 2009)
14. Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: *PLAS* (2008)
15. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Certified in-lined reference monitoring on .NET. In: *PLAS* (2006)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: *POPL* (2002)
17. G. Inc. Closure Compiler, <http://code.google.com/closure/compiler/>
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lindskov Knudsen, J. (ed.) *ECOOP* 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Marc Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auleutta, V. (eds.) *ECOOP* 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
20. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: Moped: A model checker for push-down systems, <http://www.fmi.uni-stuttgart.de/szs/tools/moped/>
21. Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: Ramalingam, G. (ed.) *APLAS* 2008. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
22. Maffeis, S., Taly, A.: Language-based isolation of untrusted Javascript. In: *CSF* (2009)
23. Maffeis, S., Taly, J.M.A.: Language-based isolation of untrusted JavaScript. In: *SP* (2010)
24. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A Layered Architecture for Detecting Malicious Behaviors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) *RAID* 2008. LNCS, vol. 5230, pp. 78–97. Springer, Heidelberg (2008)
25. McMillan, K.L.: Applications of Craig Interpolants in Model Checking. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS* 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)
26. Meyerovich, L., Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In: *SP* (2010)
27. Saxena, P., Akhawe, D., Hanna, S., McCamant, S., Mao, F., Song, D.: A symbolic execution framework for JavaScript. In: *SP* (2010)
28. Schneider, F.B.: Enforceable security policies. *TISSEC* 3 (February 2000)
29. Sridhar, M., Hamlen, K.W.: Model-Checking In-Lined Reference Monitors. In: Barthe, G., Hermenegildo, M. (eds.) *VMCAI* 2010. LNCS, vol. 5944, pp. 312–327. Springer, Heidelberg (2010)
30. Yu, D., Chander, A., Islam, N., Serikov, I.: JavaScript instrumentation for browser security. In: *POPL* (2007)

Automatic Quantification of Cache Side-Channels

Boris Köpf¹, Laurent Mauborgne¹, and Martín Ochoa^{2,3}

¹ IMDEA Software Institute, Spain

² Siemens AG, Germany

³ TU Dortmund, Germany

{boris.koepf,laurent.mauborgne}@imdea.org,
martin.ochoa@cs.tu-dortmund.de

Abstract. The latency gap between caches and main memory has been successfully exploited for recovering sensitive input to programs, such as cryptographic keys from implementation of AES and RSA. So far, there are no practical general-purpose countermeasures against this threat. In this paper we propose a novel method for automatically deriving upper bounds on the amount of information about the input that an adversary can extract from a program by observing the CPU’s cache behavior. At the heart of our approach is a novel technique for efficient counting of concretizations of abstract cache states that enables us to connect state-of-the-art techniques for static cache analysis and quantitative information-flow. We implement our counting procedure on top of the AbsInt TimingExplorer, one of the most advanced engines for static cache analysis. We use our tool to perform a case study where we derive upper bounds on the cache leakage of a 128-bit AES executable on an ARM processor. We also analyze this implementation with a commonly suggested (but until now heuristic) countermeasure applied, obtaining a formal account of the corresponding increase in security.

1 Introduction

Many modern computer architectures use caches to bridge the latency gap between the CPU and main memory. On today’s architectures, an access to the main memory (i.e. a cache miss) may imply an overhead of hundreds of CPU cycles w.r.t. an access to the cache (cache hit). While the use of caches is beneficial for performance reasons, it can have negative effects on security: An observer who can measure the time of memory lookups can see whether a lookup is a cache hit or miss, thereby learning partial information about the state of the cache. This partial information has been used for extracting cryptographic keys from implementations of AES [12, 22, 35], RSA [37], and DSA [6]. In particular AES is vulnerable to such cache attacks, because most high-speed software implementations make heavy use of look-up tables. Cache attacks are the most effective known attacks against AES and allow to recover keys within minutes [22].

A number of countermeasures have been proposed against cache attacks. They can be roughly put in two classes: (1) Avoiding the use of caches for sensitive

computations. This can be achieved, e.g. by using dedicated hardware implementations (For example, recent Intel processors offer support for AES), or by side-stepping the use of caches in software implementations [25]. Both solutions obviously defeat cache attacks; however they are not applicable to arbitrary programs, e.g. due to lack of available hardware support, or for reasons of performance. (2) Mitigation strategies for eliminating attack vectors and reducing leakage. Proposals include disabling high-resolution timers, hardening of schedulers [22], and preloading [12,35] of tables. Such strategies are implemented, e.g. in the OpenSSL 1.0 [5] version of AES, however, their effectiveness is highly dependent on the operating system and the CPU. Without considering/modeling all implementation details, such mitigation strategies necessarily remain heuristic. In summary, there is no general-purpose countermeasure against cache attacks that is backed-up by mathematical proof.

In this paper, we propose a novel method for establishing formal security guarantees against cache-attacks that is applicable to arbitrary programs and a wide range of embedded platforms. The guarantees we obtain are upper bounds on the amount of information about the input that an adversary can extract by observing which memory locations are present in the CPU’s cache after execution of the program; they are based on the actual program binary and a concrete processor model and can be derived entirely automatically. At the heart of our approach is a novel technique for effective counting of concretizations of abstract states that enables us to connect state-of-the-art techniques for static cache analysis and quantitative information-flow analysis.

Technically, we build on prior work on static cache analysis [20] that was primarily used for the estimation of worst-case execution time by abstract interpretation [17]. We also leverage techniques from quantitative-information-flow analysis that enable establishing bounds for the amount of information that a program leaks about its input. One key observation is that (an upper bound on) the number of reachable states of a program corresponds to (an upper bound on) the number of leaked bits [30,40]. Such upper bounds can be obtained by computing super-sets of the set of reachable states by abstract interpretation, and by determining their sizes [30].

We develop a novel technique for counting the number of cache states represented by the abstract states of the static cache analyses described above. We implement this technique in a counting engine which we connect to AbsInt’s a^3 [1], the state-of-the-art tool for static cache analysis. a^3 efficiently analyzes binary code based on accurate models of several modern embedded processors with a wide range of cache types (e.g. data caches, instruction caches, or mixed) and replacement strategies. Using this tool-chain, we perform an analysis of a binary implementation of 128-bit AES from the PolarSSL library [3], based on a 32-bit ARM processor with a 4-way set associative data cache with LRU replacement strategy. We analyze this implementation with and without the preloading countermeasure applied, with different cache sizes, and for two different adversary models, obtaining the following results.

Without preloading, the derived upper bounds for the leakage (about the payload and the key) in one execution exceed the size of the key and are hence too large for practical use. With preloading and a powerful adversary model, however, the derived bounds drop to values ranging from 55 to 1 bits, for cache sizes ranging from 16KB to 128KB. With a less powerful but realistic adversary model, the bounds drop even further to ranges from 6 to 0 bits, yielding strong security guarantees. This case study shows that the automated, formal security analysis of realistic cryptosystems and accurate real processor models is in fact feasible.

In summary, our contributions are threefold. Conceptually, we show how state-of-the-art tools for quantitative information-flow analysis and static cache analysis can be combined for quantifying cache side-channels. Technically, we develop and implement novel methods for counting abstract cache states. Practically, we perform a formal cache-analysis of a binary AES 128 implementation on a realistic processor model.

2 Preliminaries

In this section we revisit concepts from quantitative information-flow analysis. In particular, we introduce measures of confidentiality based on information theory in Section 2.1, and we present techniques for their approximation in Section 2.2.

2.1 Quantifying Information Leaks

A (deterministic) *channel* is a function $C: S \rightarrow O$ mapping a finite set of secrets S to a finite set of observations O . We characterize the security of a channel in terms of the difficulty of guessing the secret input from the observation. This difficulty can be captured using information-theoretic entropy, where different notions of entropy correspond to different kinds of guessing. In this paper, we focus on min-entropy as a measure, because it is associated with strong security guarantees [40].

Formally, we model the choice of a secret input by a random variable X with range $\text{ran}(X) = S$ and the corresponding observation by a random variable Y with $\text{ran}(Y) = O$. The dependency between X and Y is formalized as a conditional probability distribution $P_{Y|X}$ with $P_{Y|X}(o, s) = 1$ if $C(s) = o$, and 0 otherwise. We consider an adversary that wants to determine the value of X from the value of Y , where we assume that X is distributed according to P_X . The adversary's a priori uncertainty about X is given by the *min-entropy* [39]

$$H_\infty(X) = -\log_2 \max_s P_X(s)$$

of X , which captures the probability of correctly guessing the secret in one shot. The adversary's a posteriori uncertainty is given by the *conditional min-entropy* $H_\infty(X|Y)$, which is defined by

$$H_\infty(X|Y) = -\log_2 \sum_o P_Y(o) \max_s P_{X|Y}(s, o)$$

and captures the probability of guessing the value of X in one shot when the value of Y is known.

The (*min-entropy*) leakage L of a channel with respect to the input distribution P_X is the reduction in uncertainty about X when Y is observed,

$$L = H_\infty(X) - H_\infty(X|Y),$$

and is the logarithm of the factor by which the probability of guessing the secret is reduced by the observation. Note that L is not a property of the channel alone as it also depends on P_X . We eliminate this dependency as follows.

Definition 1 (Maximal Leakage). *The maximal leakage ML of a channel C is the maximal reduction in uncertainty about X when Y is observed*

$$ML(C) = \max_{P_X}(H_\infty(X) - H_\infty(X|Y)),$$

where the maximum is taken over all possible input distributions.

For computing an upper bound for the maximal leakage of a deterministic channel, it suffices to compute the size of the range of C . While these bounds can be coarse in general, they are tight for uniformly distributed input.

Lemma 1.

$$ML(C) \leq \log_2 |C(S)|,$$

where equality holds for uniformly distributed P_X .

Proof. The maximal leakage of a (probabilistic) channel specified by the distribution $P_{Y|X}$ can be computed by $ML(P_{Y|X}) = \log_2 \sum_o \max_s P_{Y|X}(o, s)$, where the maximum is assumed (e.g.) for uniformly distributed input [13,31]. For deterministic channels, the number of non-zero (hence 1) summands matches $|C(S)|$.

2.2 Static Analysis of Channels

In this paper we consider channels of *programs*, i.e. those that are given by the semantics of (deterministic, terminating) programs. In this setting, the set of secrets is a part of the initial state of the program, and the set of observables is a part of the final state of the program. Due to Lemma 1, computing upper bounds on the maximal leakage of a program can be done by determining the set of final states of the program. Computing this set from the program code requires computation of a fixed-point and is not guaranteed to terminate for programs over unbounded state-spaces. Abstract interpretation [17] overcomes this fundamental problem by resorting to an approximation of the state-space and the transition relation. By choosing an adequate approximation one can enforce termination of the fixed-point computation after a finite number of steps. The soundness of the analysis follows from the soundness of the abstract domain, which is expressed in terms of a *concretization function* (denoted γ) relating elements of the abstract domain to concrete properties of the program, ordered by implication.

For the purpose of this paper, we define soundness with respect to a channel, i.e., we will use a concretization function mapping to sets of observables (where implication corresponds to set inclusion).

Definition 2. An abstract element t^\sharp is sound for a concretization function γ with respect to a channel $C : S \rightarrow O$ if and only if $C(S) \subseteq \gamma(t^\sharp)$.

The following theorem is an immediate consequence from Lemma 1; it states that a counting procedure for $\gamma(t^\sharp)$ can be used for deriving upper bounds on the amount of information leaked by C .

Theorem 1. Let t^\sharp be sound for γ with respect to C . Then

$$ML(C) \leq \log_2 |\gamma(t^\sharp)| .$$

For a more detailed account of the connection between abstract interpretation and quantitative information-flow, see [30].

3 Cache Channels

In this section, we define channels corresponding to two adversary models that can only observe cache properties. We also revisit two abstract domains for reasoning about cache-states and show how they relate to those channels. We begin with a primer on caching.

3.1 Caches

Typical caches work as follows. The main memory is partitioned into *blocks* of size B that are referenced using locations *loc*. A cache consists of a number of *sets*, each containing a fixed number of *lines* that can each store one memory block. The size A of the cache sets is called the *associativity* of the cache. Each memory block can reside in exactly one cache set, which is determined by the block's location. We can formally define a single *cache set* as a mapping

$$t: \{1, \dots, A\} \rightarrow loc \cup \{\perp\} ,$$

from line numbers to locations, where \perp represents an empty line. The mapping t is injective, which captures that a memory block is stored in at most one line. A *cache* is a tuple of independent cache sets. For simplicity of presentation, we focus on single cache sets throughout the paper, except for the case study in Section 5.

What happens when a memory block is requested depends on the *replacement strategy*. Here we focus on the LRU (Least Recently Used) strategy, which is used e.g. in the Pentium I processor. With LRU, each cache set forms a queue. When a memory block is requested, it is appended to the head of the queue. If the block was already stored in the cache (cache hit), it is removed from its original position; if not (cache miss), it is fetched from main memory. Due to the queue

structure of sets, memory blocks *age* when other blocks are looked up, i.e. they move towards the tail of the queue and (due to the fixed length of the queue) are eventually removed. For a formalization of the LRU set update function see [20]. For a formalization of alternative update functions, such as FIFO (First In First Out) see [21,38]. Depending on the concrete processor model, data and instructions are processed using dedicated caches or a common one [20]. Unless mentioned otherwise (e.g. in the experiments on AES), our results hold for any cache analysis that is sound.

3.2 Two Adversary Models Observing the Cache

We consider a scenario where multiple processes share a common CPU. We assume that one of these processes is adversarial and tries to infer information about the computations of a victim process by inspecting the cache after termination. We distinguish between two adversaries Adv_{prec} and Adv_{prob} . Both adversaries can modify the initial state of the cache with memories in their virtual memory space, which we assume is not shared between processes, but they differ in their ability of observing the final cache state:

Adv_{prec} : This adversary can observe which memory blocks are contained in the cache at the end of the victim's computation.

Adv_{prob} : This adversary can observe which blocks of his virtual memory space are contained in the cache after the victim's computation.

Note that neither adversary can observe the actual data that is stored in the victim's memory blocks that reside in the cache. The channel corresponding to the adversary Adv_{prec} simply maps the victim's input to the corresponding final cache state. The channel corresponding to Adv_{prob} can be seen as an abstraction of the channel corresponding to Adv_{prec} , as it can be described as the composition of the channel of Adv_{prec} with a function *blur* that maps all memory blocks not belonging to the adversary's virtual memory space to one indistinguishable element. Adv_{prob} corresponds to the adversaries encountered in synchronous "prime and probe" attacks [35], which observe the cache-state by performing accesses to different locations and use timing measurements to distinguish whether they are contained in the cache or not.

Considering that our adversary models allow some choice of the initial state, they formally define families of channels that are indexed by the adversarially chosen part of the initial cache. To give an upper bound on the leakage of all channels in those families we would need relational information, which is not supported by the existing cache analysis tools. One possible solution is to consider an abstract initial state approximating all possible adversary choices, which leads to imprecision in the analysis. In the particular case of a LRU replacement strategy, we can use the following property:

Proposition 1. *For caches with LRU strategy, the leakage to Adv_{prec} (Adv_{prob}) w.r.t. any initial cache state containing only memory locations from the adversary's memory space is upper-bounded by the leakage to Adv_{prec} (Adv_{prob}) w.r.t. an empty initial cache state.*

This result follows from the following observation: for each initial cache state containing locations disjoint from the victim’s memory space, the first i lines of the final cache state will contain the locations accessed by the victim, and the remaining lines will contain the first $A - i$ locations of the initial state shifted to the right, where i depends on that particular run of the victim. That is, modulo the adversarial locations, the number of possible final cache states corresponding to an empty initial state matches the number of final cache states corresponding to an initial state that does not contain locations from the victim’s memory space. The assertion then follows immediately from Theorem 1. Proposition 1 will be useful in our case study, since the analysis we use provides a more accurate final state when run with an empty initial cache.

3.3 Abstract Domains for Cache Analysis

Ferdinand et al. [20] propose abstract interpretation techniques for cache analysis and prove their soundness with respect to reachability of cache states, which corresponds to soundness w.r.t the channel of Adv_{prec} according to Definition 2. In particular, they present two abstract domains for cache-states: The first domain corresponds to a *may*-analysis and represents the set of memory locations that possibly reside in the cache. The second domain corresponds to a *must*-analysis and represents the set of memory locations that are definitely in the cache. In both cases, an abstract cache set is represented as a function

$$t^\sharp: \{1, \dots, A\} \rightarrow 2^{loc}$$

mapping set positions to sets of memory locations, where $t^\sharp(i) \cap t^\sharp(j) = \emptyset$ whenever $i \neq j$. In the following we will use t_1^\sharp and t_2^\sharp for abstract sets corresponding to the *may* and *must* analysis respectively.

For the *may* analysis, the concretization function γ^{\cup} is defined by

$$\gamma^{\cup}(t_1^\sharp) = \{t \mid \forall i \in \{1, \dots, A\}: t(i) \in \bigcup_{j=1}^i t_1^\sharp(j) \cup \{\perp\}\}$$

This definition implies that each location that appears in the concrete state appears also in the abstract state, and the position in the abstract state is a lower bound for the position in the concrete.

For the *must* analysis, the concretization function γ^{\cap} is defined by

$$\gamma^{\cap}(t_2^\sharp) = \{t \mid \forall i \in \{1, \dots, A\}: t_2^\sharp(i) \subseteq \bigcup_{j=1}^i \{t(j)\}\}$$

This definition implies that each location that appears in the abstract state is required to appear in the concrete, and its position in the abstract is an upper bound for its position in the concrete.

Example 1. Consider the following program running on a 4-way fully associative (i.e. only one set) data cache where $\dots x \dots$ stands for an instruction that references location x , and let e, a, b are pairwise distinct locations.

```
if ... e ... then ... a ... else ... b ...
```

With an empty initial abstract cache before execution, the may- and must-analyses return the following abstract final states:

$$t_1^\# = [\{a, b\}, \{e\}, \{\}, \{\}] \quad t_2^\# = [\{\}, \{e\}, \{\}, \{\}]$$

Both $\gamma^U(t_1^\#)$ and $\gamma^M(t_2^\#)$ contain the two reachable states $[a, e, \perp, \perp]$ and $[b, e, \perp, \perp]$ (which is due to the soundness of the analyses) but also unreachable states such as $[\perp, e, a, b]$ (which is due to the imprecision of the analyses). In particular, states in which empty cache lines are followed by non-empty cache lines are artifacts of the abstraction, i.e. they cannot occur according to the concrete cache semantics from [20]. More precisely, we have

$$\forall i, j \in \{1, \dots, A\} : t(i) = \perp \wedge j > i \implies t(j) = \perp . \quad (1)$$

It is hence sufficient to consider only the concrete states that also satisfy (1), which enables us to derive tighter bounds in Section 4. For simplicity of notation we will implicitly assume that (1) is part of the definition of γ^U and γ^M .

To obtain the channel corresponding to the adversary model Adv_{prob} , we just need to apply *blur* to the concretization of the must and may cache analysis, which is equivalent to first applying *blur* to the sets appearing in the abstract elements and then concretizing.

4 Counting Cache States

We have introduced channels corresponding to two adversaries, together with sound abstract interpretations. The final step needed for obtaining an automatic quantitative information-flow analysis from Theorem 1 are algorithms for counting the concretizations of the abstract cache states presented in Section 3.3, which we present next. As before, we restrict our presentation to single cache sets. Counting concretizations of caches with multiple sets can be done by taking the product of the number of concretizations of each set.

4.1 Concrete States Respecting *may*

We begin by deriving a formula for counting the concretizations of an abstract may-state $t_1^\#$. To this end, let $n_i = |t_1^\#(i)|$, $n_i^* = \sum_{j=1}^i n_j$, for all $i \in \{1, \dots, A\}$ and $n^* = n_A^*$. The definition of $\gamma^U(t_1^\#)$ informally states that, when reading the content of $t_1^\#$ and $t \in \gamma^U(t_1^\#)$ from head to tail in lockstep, each non-empty line in t has appeared in the same or a previous line of $t_1^\#$. That is, for filling line k of t there are n_k^* possibilities, of which $k - 1$ are already used for filling lines

$1, \dots, k - 1$. The number of concrete states with a fixed number i of non-empty lines is hence given by

$$\prod_{k=1}^i (n_k^* - (k - 1)) \quad (2)$$

As the definition of γ^\cup does not put a lower bound on the number i of nonempty lines, we need to consider all $i \in \{1, \dots, A\}$. We obtain the following explicit formula for the number of concretizations of t_1^\sharp .

Proposition 2 (Counting May).

$$|\gamma^\cup(t_1^\sharp)| = \sum_{i=0}^A \prod_{k=1}^i (n_k^* - (k - 1))$$

Example 2. When applied to the abstract may-state $t_1^\sharp = [\{a, b\}, \{e\}, \{\}, \{\}]$ obtained from the analysis of the program in Example 1 we obtain $|\gamma^\cup(t_1^\sharp)| = 11$, which illustrates that the bounds obtained by Proposition 2 can be coarse.

4.2 Concrete States Respecting *must*

For counting the concretizations of an abstract must-state t_2^\sharp , let $m_i = |t_2^\sharp(i)|$, $m_i^* = \sum_{j=1}^i m_j$, for all $i \in \{1, \dots, A\}$ and $m^* = m_A^*$. The definition of γ^\cap informally states that when reading the lines of an abstract state t_2^\sharp and a concrete state $t \in \gamma^\cap(t_2^\sharp)$ from head to tail in lockstep, each element of t_2^\sharp has already appeared in the same or a previous line of t . More precisely, the m_j elements contained in line j of t_2^\sharp appear in lines $1, \dots, j$ of t , of which m_{j-1}^* are already occupied by the must-constraints of lines $1, \dots, j - 1$. This leaves $\binom{j-m_{j-1}^*}{m_j} m_j!$ possibilities for placing the elements of $t_2^\sharp(j)$, which amounts to a total of

$$\prod_{j=1}^A \binom{j - m_{j-1}^*}{m_j} m_j! \quad (3)$$

possibilities for placing all elements in t_2^\sharp . However, notice that $m^* \leq A$ is possible, i.e. must-constraints can leave cache lines unspecified. The number of possibilities for filling those unspecified lines is

$$\prod_{k=m^*+1}^A (\ell - (k - 1)), \quad (4)$$

where $\ell = |\text{loc}|$ is the number of possible memory locations.

Finally, observe that (3) and (4) count concrete states in which each line is filled. However, the definition γ^\cap only mandates that at least m^* lines of each concrete state be filled. We account for this by introducing a variable i that ranges from m^* to A . We modify (3) by choosing from $\min(i, j)$ instead of j positions¹ and we modify (4) by replacing the upper bound by i . This yields the following for explicit formula for the number of concretizations of t_2^\sharp .

¹ The index j still needs to go up to A in order to collect all constraints.

Proposition 3 (Counting Must).

$$|\gamma^\cap(t_2^\#)| = \sum_{i=m^*}^A \left(\prod_{j=1}^A \binom{\min(i, j) - m_{j-1}^*}{m_j} m_j! \prod_{k=m^*+1}^i (\ell - (k-1)) \right)$$

Example 3. When applied to the must-state $t_2^\# = [\{\}, \{e\}, \{\}, \{\}]$ and a set of locations $loc = \{a, b, c, d, e\}$, Proposition 3 yields a number of 81 concretizations of $t_2^\#$. This over-approximation stems from the fact that the abstract state requires only the containment of e and that the rest of the lines can be chosen from loc . We next tackle this imprecision by considering the intersection of may and must.

4.3 Concrete States Respecting *must* and *may*

For computing the number of concrete states respecting both $t_2^\#$ and $t_1^\#$ we reuse the notation introduced in Sections 4.1 and 4.2. As in Section 4.2 we use (3) for counting the cache lines constrained by the must-information. However, instead of filling the unconstrained lines with *all* possible memory locations, we now choose only from the lines specified by the may-information. The counting is similar to equation (2), the difference being that, as in (1), the product starts with $k = m^* + 1$ because the content of m^* lines is already fixed by the must-constraints. The key difference to (4) is that now we pick only from at most n_k^* lines instead of ℓ lines. We obtain the following proposition.

Proposition 4 (Counting May and Must).

$$|\gamma^\cup(t_1^\#) \cap \gamma^\cap(t_2^\#)| \leq \sum_{i=m^*}^A \left(\prod_{j=1}^A \binom{\min(i, j) - m_{j-1}^*}{m_j} m_j! \prod_{k=m^*+1}^i (n_k^* - (k-1)) \right)$$

Two comments are in order. First, notice that the inequality in Proposition 4 stems from the fact that the lines unconstrained by the must-information may be located at positions $j < k$. Using the constraint n_j^* instead of n_k^* would lead to tighter bounds, however, an explicit formula for this case remains elusive. Second, observe that the rightmost product is always non-negative. For this it is sufficient to prove that the first factor $n_{m^*+1}^* - m^*$ is non-negative, because the value of subsequent factors decreases by at most 1. Assume that $n_{m^*+1}^* - m^* < 0$ (and hence $n_{m^*}^* < m^*$). By (1), $n_j^* < j$ implies that line j is empty for all concrete states, which for $j = m^*$ contradicts the requirement that all states contain at least m^* lines.

Example 4. When applied to the abstract cache states $t_1^\# = [\{a, b\}, \{e\}, \{\}, \{\}]$ and $t_2^\# = [\{\}, \{e\}, \{\}, \{\}]$ from Example 1, Proposition 4 delivers a total of 9 concrete states.

It is easy to see that the expression in Proposition 4 can be evaluated in time $O(A^3)$ because both the factorial and n_i^* can be computed in linear time

and they are nested in two loops of length at most A . Although efficient, an approximation using Proposition 4 can be coarse: In Example 4 we computed a bound of 9 states, although (as is easily verified manually) there are only 4 concrete states respecting the constraints of both abstract states. We have developed more accurate (but more complex) variants of Proposition 4 that yield the exact bounds for this example, however, they are also not tight in general.

In the absence of a closed expression for the exact number of concrete states, one can proceed by enumerating the set of all concrete states respecting may, and filtering out those not respecting must (see [29] for an implementation). The price to pay for this brute-force approach is a worst-case time complexity of $O(A!)$, e.g. if there are no must-constraints and the first location of the abstract may-state contains A or more locations. This is not a limitation for the small associativities often encountered in practice ($A = 2$ or $A = 4$), however, for fully associative caches in which A equals the total number of lines of the cache, the approximation given by Proposition 4 is the more adequate tool.

4.4 Counting for Probing Adversaries

For counting the possible observations of Adv_{prob} for arbitrary replacement strategies, we can apply the techniques presented above to previously blurred abstract states. For the case of a LRU strategy, we obtain the following better bounds.

Proposition 5. *The number of observations Adv_{prob} can make is bounded by*

$$\min(n^*, A) - m^* + 1$$

The assertion follows from the fact that, after the computation, each cache set will first contain the victim's locations (which Adv_{prob} cannot distinguish), and then a fixed sequence of locations from the adversary's virtual memory whose length only depends on the number of the victim's blocks. I.e., when starting from an empty cache set, the adversary can only observe the length of the final cache set. This size is at least m^* (because at least that number of lines must be filled), and at most $\min(n^*, A)$. The additional 1 accounts for the empty state.

5 Case Study

In this section we report on a case-study where we use the methods developed in this paper for analyzing the cache side-channel of a widely used AES implementation on a realistic processor model with different cache configurations.

5.1 Tool Support

We have implemented a tool for the static quantification of cache side-channels, based on the development presented in this paper. Its building blocks are the AbsInt a^3 for static cache analysis, and a novel counting engine for cache-states.

Static analyzer. The AbsInt a^3 [1] is a suite of industrial-strength tools for the static analysis of embedded systems. In particular, a^3 comprises tools (called aiT and TimingExplorer) for the estimation of worst-case execution times based on [20]. The tools cover a wide range of CPUs, such as ERC32, M68020, LEON3 and several PowerPC models (aiT), as well as CPU models with freely configurable LRU cache (TimingExplorer). We base our implementation on the TimingExplorer for reasons of flexibility. The TimingExplorer receives as input a program binary and a cache configuration and delivers as output a control flow graph in which each (assembly-level) instruction is annotated by the corresponding abstract *may* and *must* states. We extract the annotations of the final state of the program, and provide them as input to the counting engine.

Counting engine. Our counting engine determines the number of concretizations of abstract cache states according to the development in Section 4. Our language of choice is Haskell [4], because it allows for a concise representation of sums, products, and enumerations using list comprehensions. A detailed description of the routines for exact counting can be found in the extended version of the paper [29].

5.2 Target Implementations

Code. We analyze the implementation of 128 bit AES encryption from the PolarSSL library [3], a lightweight crypto suite for embedded platforms. As is standard for software implementations of AES, the code consists of single loop (corresponding to the rounds of AES) in which heavy table lookups are performed to indices computed using bit-shifting and masking. We also analyze a modified version of this implementation, where we add a loop that loads the entire lookup table into the cache before encryption. This preloading has been suggested as countermeasure against cache attacks because, intuitively, all lookups during encryption will hit the cache.

Platform. We compile the AES C source code into a binary for the ARM7TDMI CPU [2]. Although the original ARM7TDMI does *not* have any caches, the AbsInt TimingExplorer supports this CPU with the possibility of specifying arbitrary configurations of data/instruction/mixed caches with LRU strategy. For our experiments we use data caches with sizes of 16-128 KB, associativity of 4 ways, and a line size of 32 Bytes, which are common configurations in practice.

5.3 Improving Precision by Partitioning

The TimingExplorer can be very precise for simple expressions, but loses precision when analyzing array lookups to non-constant indexes. This source of imprecision is well-known in static analysis, and abstract interpretation offers techniques to regain precision, such as abstract domains specialized for arrays [18], or automatic refinement of transfer functions. For our analysis, we use results on

trace partitioning [32], which consists in performing the analysis on a partition of all possible runs of a program, each partition yielding more precise results.

We have implemented a simple trace partitioning strategy using program transformations that do not modify the data cache (which is crucial for the soundness of our approach). For each access to the look-up table, we introduce conditionals on the index, where each branch corresponds to one memory block, and we perform the table access in all branches. As the conditionals cover all possible index values for the table access, we add one memory access to the index before the actual table look-up, which does not change the cache state for an LRU cache strategy, since the indices have to be fetched before accessing the table anyway. An example of the AES code with trace partitioning can be found in the extended version of this paper [29].

Note that the same increase in precision could be achieved without program transformation if the trace partitioning were implemented at the level of the abstract interpreter, which would also allow us to consider instruction caches and cache strategies beyond LRU. Given that the TimingExplorer is closed-source, we opted for partitioning by code transformation.

5.4 Results and Security Interpretation

The results of our analysis with respect to the adversary Adv_{prec} are depicted in Figure 11. For AES without preloading of tables, the bounds we obtained exceed 160 bits for all cache sizes. For secret keys of only 128 bits, they are not precise enough for implying meaningful security guarantees. With preloading, however, those bounds drop down to 55 bits for caches sizes of 16KB and to only 1 bit for sizes of 128KB, showing that only a small (in the 128KB case) fraction of the key bits can leak in one execution. The results of our analysis with respect to the (less powerful, but more realistic) adversary Adv_{prob} are depicted in Figure 12. As for Adv_{prec} , the bounds obtained without preloading exceed the size of the secret key. With preloading, however, they remain below 6 bits and even drop to 0 bits for caches of 128KB, giving a formal proof of noninterference for this implementation and platform.

Notice that the leakage bounds we derive hold for single executions. For the case of zero leakage they trivially extend to bounds for multiple executions and immediately imply strong security guarantees. For the case of non-zero leakage, the bounds can add up when repeatedly running the victim process with a fixed key and varying payload, leading to a decrease in security guarantees. See Section 7 for possible solutions to this problem.

6 Prior Art

Timing attacks against cryptosystems date back to [26]. They can be divided into those exploiting timing variations due to control-flow [14,26] and those exploiting timing variations of the execution platform, e.g. due to caches [7,9,12,35,36,37], or branch prediction units [8]. In this paper we focus solely on caching.

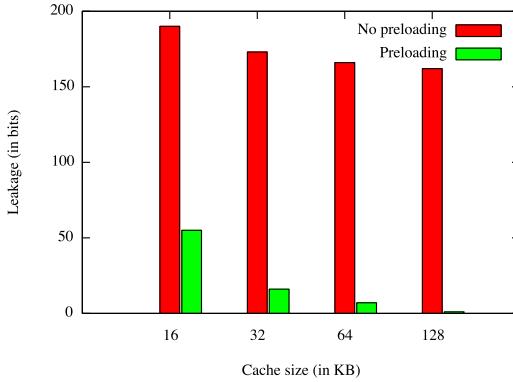


Fig. 1. Upper bounds for the maximal leakage w.r.t. the adversary Adv_{prec} and a 4-way set associative cache with 32B lines of sizes 16KB-128KB

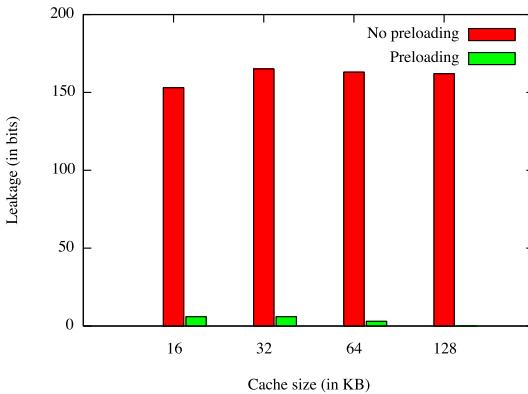


Fig. 2. Upper bounds for the maximal leakage w.r.t. the adversary Adv_{prob} and a 4-way set associative cache with 32B lines of sizes 16KB-128KB

The literature on cache attacks is stratified according to a variety of different adversary models: In *time-driven attacks* [9, 12] the adversary can observe the overall execution time of the victim process and estimate the overall number of cache hits and misses. In *trace-driven attacks* [7] the adversary can observe whether a cache hit or miss occurs, for every single memory access of the victim process. In *access-driven attacks* [35, 37] the adversary can probe the cache either during computation (*asynchronous* attacks) or after completion (*synchronous* attacks) of the victim’s computation, giving him partial information about the memory locations accessed by the victim. Finally, some attacks assume that the adversary can choose the cache state before execution of the victim process [35], whereas others only require that the cache does not contain the locations that are looked-up by the victim during execution [9]. The information-theoretic bounds we derive hold for single executions of synchronous access-driven adversaries,

where we consider initial states that do not contain the victim’s data. The derivation of bounds for alternative adversary models is left future work.

A number of mitigation techniques have been proposed to counter cache attacks. Examples include coding guidelines [16] for thwarting cache attacks on x86 CPUs, or novel cache-architectures that are more resistant to cache attacks [42]. One commonly proposed mitigation technique is preloading of tables [12, 35]. However, as first observed by [12], it is a non-trivial issue to establish the efficacy of this countermeasure. As [35] comments:

[...], it should be ensured that the table elements are not evicted by the encryption itself, by accesses to the stack, inputs or outputs. Ensuring this is a delicate architecture-dependent affair [...].”

The methods developed in this paper enable us to automatically and formally deal with these delicate affairs based on an accurate model of the CPU.

For the case of AES, there are efficient software implementations that avoid the use of data caches by bit-slicing, and achieve competitive performance by relying on SIMD (Single Instruction, Multiple Data) support [25]. Furthermore, a model for statistical estimation of the effectiveness of AES cache attacks based on sizes of cache lines and lookup tables has been presented in [41]. For programs beyond AES that are not efficiently implementable using bit-slicing, our analysis technique enables the derivation of formal assertions about their leakage, based on the actual program semantics and accurate models of the CPU.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [15], where the automation by reduction to counting appears in [11, 24, 33, 34], and the connection to abstract interpretation in [30]. Prior applications of QIF to side-channels in cryptosystems [27, 28, 31] are limited to stateless systems. For the analysis of caches, we rely on the abstract domains from [20] and their implementation in the AbsInt TimingExplorer [1]. Finally, our work goes beyond language-based approaches that consider caching [10, 23] in that we rely on more realistic models of caches and aim for more permissive, quantitative guarantees.

7 Conclusions and Future Work

We have shown that cache side-channels can be automatically quantified. For this, we have leveraged powerful tools for static cache analysis and quantitative information-flow analysis, which we connect using novel techniques for counting the concretizations of abstract cache states. We have demonstrated the practicality of our approach by deriving information-theoretic security guarantees for an off-the-shelf implementation of 128-bit AES (with and without a commonly suggested countermeasure) on a realistic model of an embedded CPU.

Our prime target for future work is to derive security guarantees that hold for multiple executions of the victim process. One possibility to achieve this is to extend static cache analysis along the lines of [27]. Another possibility is to employ leakage-resilient cryptosystems [19], where our work can be used for bounding the range of the leakage functions. Further avenues are to extend our quantification to cater for alternative adversary models, such as asynchronous,

trace-based, and timing-based. Progress along these lines will enable the automatic derivation of formal, quantitative security guarantees for a larger class of relevant attack scenarios.

Acknowledgments. We thank Reinhard Wilhelm for pointing us to his group's work on static cache analysis, the AbsInt team for their friendly support, Stephan Max for his help with the TimingExplorer, and Daniel Bernstein, Pierre Ganty, and Andrew Myers for helpful feedback and suggestions. This work was partially funded by European projects FP7-256980 NESSoS and FP7-229599 AMAROUT, and by the MoDelSec project of the DFG priority programme 1496 RS³.

References

1. AbsInt aiT Worst-Case Execution Time Analyzers, <http://www.absint.com/a3/>
2. Arm7tdmi datasheet, <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0234b/DDI0234.pdf>
3. PolarSSL, <http://polarssl.org/>
4. The Haskell Programming Language, <http://www.haskell.org/>
5. The Open Source toolkit for SSL/TSL, <http://www.openssl.org/>
6. Aciçmez, O., Brumley, B.B., Grabher, P.: New Results on Instruction Cache Attacks. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 110–124. Springer, Heidelberg (2010)
7. Aciçmez, O., Koç, Ç.K.: Trace-Driven Cache Attacks on AES (Short Paper). In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 112–121. Springer, Heidelberg (2006)
8. Aciçmez, O., Koç, Ç.K., Seifert, J.-P.: Predicting Secret Keys Via Branch Prediction. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 225–242. Springer, Heidelberg (2007)
9. Aciçmez, O., Schindler, W., Koç, Ç.K.: Cache Based Remote Timing Attack on the AES. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 271–286. Springer, Heidelberg (2007)
10. Agat, J.: Transforming out Timing Leaks. In: POPL, pp. 40–53. ACM (2000)
11. Backes, M., Köpf, B., Rybalchenko, A.: Automatic Discovery and Quantification of Information Leaks. In: SSP, pp. 141–153. IEEE (2009)
12. Bernstein, D.J.: Cache-timing attacks on AES. Technical report (2005)
13. Braun, C., Chatzikokolakis, K., Palamidessi, C.: Quantitative notions of leakage for one-try attacks. In: MFPS. ENTCS, vol. 249, pp. 75–91. Elsevier (2009)
14. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks 48(5), 701–716 (2005)
15. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. JCS 15(3), 321–371 (2007)
16. Coppens, B., Verbauwheide, I., Bosschere, K.D., Sutter, B.D.: Practical mitigations for timing-based side-channel attacks on modern x86 processors. In: SSP, pp. 45–60. IEEE (2009)
17. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In: POPL, pp. 238–252 (1977)
18. Cousot, P., Cousot, R., Mauborgne, L.: A Scalable Segmented Decision Tree Abstract Domain. In: Manna, Z., Peled, D.A. (eds.) Pnueli Festschrift. LNCS, vol. 6200, pp. 72–95. Springer, Heidelberg (2010)

19. Dziembowski, S., Pietrzak, K.: Leakage-Resilient Cryptography. In: FOCS, pp. 293–302. IEEE (2008)
20. Ferdinand, C., Martin, F., Wilhelm, R., Alt, M.: Cache behavior prediction by abstract interpretation. Science of Computer Programming 35(2), 163–189 (1999)
21. Grund, D.: Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU. PhD thesis, Saarland University (2012)
22. Gullasch, D., Bangerter, E., Krenn, S.: Cache games - bringing access-based cache attacks on aes to practice. In: SSP, pp. 490–505. IEEE (2011)
23. Hedin, D., Sands, D.: Timing Aware Information Flow Security for a JavaCard-like Bytecode. ENTCS 141(1), 163–182 (2005)
24. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: ACSAC, pp. 261–269. ACM (2010)
25. Käsper, E., Schwabe, P.: Faster and Timing-Attack Resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer, Heidelberg (2009)
26. Kocher, P.: 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)
27. Köpf, B., Basin, D.: An Information-Theoretic Model for Adaptive Side-Channel Attacks. In: CCS, pp. 286–296. ACM (2007)
28. Köpf, B., Dürmuth, M.: A Provably Secure And Efficient Countermeasure Against Timing Attacks. In: CSF, pp. 324–335. IEEE (2009)
29. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. Cryptology ePrint Archive, Report 2012/034 (2012)
30. Köpf, B., Rybalchenko, A.: Approximation and Randomization for Quantitative Information-Flow Analysis. In: CSF, pp. 3–14. IEEE (2010)
31. Köpf, B., Smith, G.: Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In: CSF, pp. 44–56. IEEE (2010)
32. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
33. Meng, Z., Smith, G.: Calculating bounds on information leakage using two-bit patterns. In: PLAS. ACM (2011)
34. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: PLAS, pp. 73–85. ACM (2009)
35. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
36. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel (2002)
37. Percival, C.: Cache missing for fun and profit. In: BSDCan (2005)
38. Reineke, J.: Caches in WCET Analysis. PhD thesis, Saarland University (2008)
39. Rényi, A.: On measures of entropy and information. In: Berkeley Symp. on Mathematics, Statistics and Probability, pp. 547–561 (1961)
40. 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)
41. Tiri, K., Aciçmez, O., Neve, M., Andersen, F.: An Analytical Model for Time-Driven Cache Attacks. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 399–413. Springer, Heidelberg (2007)
42. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. In: ISCA, pp. 494–505. ACM (2007)

Secure Programming via Visibly Pushdown Safety Games

William R. Harris¹, Somesh Jha¹, and Thomas Reps^{1,2}

¹ University of Wisconsin-Madison, Madison, WI, USA

{wrharris,jha,reps}@cs.wisc.edu

² GrammaTech, Inc., Ithaca NY, USA

Abstract. Several recent operating systems provide system calls that allow an application to explicitly manage the privileges of modules with which the application interacts. Such *privilege-aware operating systems* allow a programmer to write a program that satisfies a strong security policy, even when it interacts with untrusted modules. However, it is often non-trivial to rewrite a program to correctly use the system calls to satisfy a high-level security policy. This paper concerns the *policy-weaving problem*, which is to take as input a program, a desired high-level policy for the program, and a description of how system calls affect privilege, and automatically rewrite the program to invoke the system calls so that it satisfies the policy. We present an algorithm that solves the policy-weaving problem by reducing it to finding a winning modular strategy to a visibly pushdown safety game, and applies a novel game-solving algorithm to the resulting game. Our experiments demonstrate that our algorithm can efficiently rewrite practical programs for a practical privilege-aware system.

1 Introduction

Developing practical but secure programs remains a difficult, important, and open problem. Web servers and VPN clients execute unsafe code, and yet are directly exposed to potentially malicious inputs from a network connection [24]. System utilities such as Norton Antivirus scanner [20], tcpdump, the DHCP client dhclient [23], and file utilities such as bzip, gzip, and tar [16,21,22] have contained unsafe code with well-known vulnerabilities that allow them to be compromised if an attacker can control their inputs. Once an attacker compromises any of the above programs, they can typically perform any action allowed for the user that invoked the program, because the program does not restrict the privileges with which its code executes.

Traditional operating systems provide to programs only weak primitives for managing their privileges [9,18,23,24]. As a result, if a programmer is to verify that his program is secure, he typically must first verify that the program satisfies very strong properties, such as memory safety. However, recent work [9,18,23,24] has produced new operating systems that allow programmers to develop programs that execute unsafe code but still satisfy strong properties, and

to construct such programs with significantly less effort than fully verifying the program. Such systems map each program to a set of privileges, and extend the set of system calls provided by a traditional operating system with security-specific calls (which henceforth we will call *security primitives*) that the program invokes to manage its privileges. We call such systems *privilege-aware systems*.

This paper concerns the *policy-weaving problem*, which is to take a program and a security policy that defines what privileges the program must have, and to automatically rewrite the program to correctly invoke the primitives of a privilege-aware system so that the program satisfies the policy when run on the system. The paper addresses two key challenges that arise in solving the policy-weaving problem. First, a privilege-aware system cannot allow a program to modify its privileges arbitrarily, or an untrusted module of the program could simply give itself the privileges that it requires to carry out an attack. Instead, the system allows a program to modify its privileges subject to system-specific rules. In practice, these rules are subtle and difficult to master; the developers of the Capsicum capability system reported issues in rewriting the `tcpdump` network utility to use the Capsicum primitives to satisfy a security policy, while preserving the original functionality of `tcpdump` [23].

Second, the notions of privilege often differ between privilege-aware systems, and thus so too do the primitives provided by each system, along with the rules relating privileges to primitives. The Capsicum operating system defines privileges as capabilities [23], the Decentralized Information Flow Control (DIFC) operating systems Asbestos, HiStar, and Flume [9,18,24] define privileges as the right to send information, and each provide different primitives for manipulating information-flow labels [7]. Thus, a policy-weaving algorithm for a specific system must depend on the privileges and primitives of the system, yet it is undesirable to manually construct a new policy-weaving algorithm for each privilege-aware system that has been or will yet be developed.

We address the above challenges by reducing the policy-weaving problem to finding a winning Defender strategy to a two-player safety game. Each game is played by an Attacker, who plays program instructions, and a Defender, who plays system primitives. The game accepts all sequences of instructions and primitives that violate the given policy. A winning Defender strategy never allows the Attacker to generate a play accepted by the game, and thus corresponds to a correct instrumentation of the program, which invokes primitives so that the policy is never violated. If the rules describing how a system's primitives modify privileges can be encoded as an appropriate automaton, then the game-solving algorithm can be applied to rewrite programs for the system. We argue that stack-based automata, in particular *visibly pushdown automata* (VPAs) [5], are sufficient to model the rules of practical privilege-aware systems. Furthermore, *modular* winning strategies exactly correspond to correct instrumentations of programs for such systems.

Finding a modular winning strategy to a game defined by a VPA is NP-complete. However, games resulting from policy-weaving problems are constructed as products of input automata, and a game will often have a strategy

whose structure closely matches one of the inputs. Inspired by this observation, we present a novel algorithm that, given a game, finds a modular strategy with structure similar to an additional, potentially smaller game called a *scaffold*. We show that our scaffolding algorithm generalizes two known algorithms for finding modular strategies [4, 19] — in particular, those algorithms result from using two (different) “degenerate” scaffolds, and correspond to two ends of a spectrum of algorithms that can be implemented by our algorithm. We evaluated the scaffold-based algorithm on games corresponding to policy-weaving problems for six UNIX utilities with known vulnerabilities for the Capsicum capability system, and found that it could rewrite practical programs efficiently, and that the choice in scaffold often significantly affected the performance of the algorithm.

Organization §2 motivates by example the policy-weaving problem and our game-solving algorithm. §3 defines the policy-weaving problem, and reduces the problem to solving visibly-pushdown safety games. §4 presents a novel algorithm for solving visibly pushdown safety games. §5 presents an experimental evaluation of our algorithm. §6 discusses related work.

2 Overview

In this section, we motivate the policy-weaving problem. We sketch how the policy-weaving problem can be reduced to finding a winning strategy to a class of safety games, and how the structure of games constructed from policy-weaving problems makes them amenable to our novel game-solving algorithm.

2.1 An Example Policy-Weaving Problem: Filter on MiniCap

We illustrate the policy-weaving problem using an example program **Filter** that reads information from an input channel, processes and compresses the data, and then writes the data to an output channel. **Filter** is inspired by the UNIX utilities **tcpdump** and **gzip**, which have exhibited security vulnerabilities in the past, and have previously been rewritten manually for the Capsicum privilege-aware systems [23]. The executions of **Filter** are presented as the runs of the automaton F in Fig. II(a), where each transition is labeled with a program action. Intraprocedural transitions are denoted with solid lines. Call transitions, which place their source node on a stack (see Defn. 3), are denoted with dashed lines. Return transitions, defined by the top two states of the stack, are denoted with dash-dot lines, where the transition from the top state of the stack is labeled with a 0, and the transition from the next state down on the stack is labeled with a 1. In each figure, doubled circles denote accepting states.

Filter executes a no-op instruction (**spin**) until it reads data from its designated input channel (e.g., UNIX **stdin**) (**read**), processes a segment of its input data (**proc**), and calls a compression function **Compress** (**call**). **Compress** first opens and reads a configuration file (**cnfg**), compresses its input data (**cmp**),

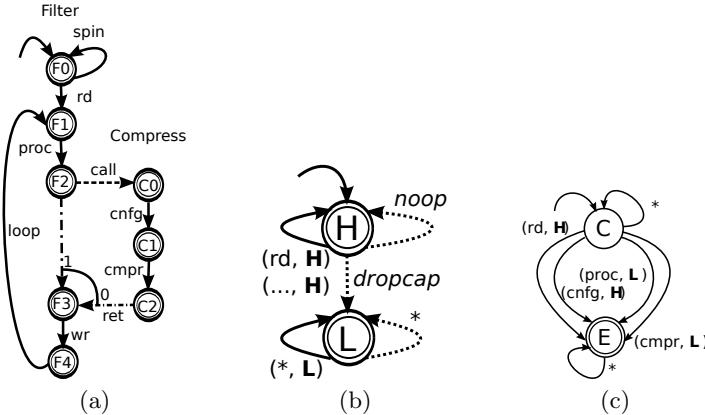


Fig. 1. Automata models of (a) the program **Filter** (**F**), (b) **Filter**'s MiniCap monitor (**M**), and (c) **Filter**'s policy for MiniCap (**Pol**). Executions of **Filter** are the runs of the automaton in (a). **Filter**'s MiniCap monitor allows sequences of privilege-instruction pairs and primitives accepted by the automaton in (b). **Filter**'s policy allows all sequences of privilege-instructions pairs accepted by the automaton in (c). Notation is explained in §2.1

and returns the result (**ret**). After **Compress** returns, **Filter** writes the data to its designated output channel (e.g., UNIX `stdout`) (**wr**), and loops to read another segment of data (**loop**).

Unfortunately, in practice, much of the code executed by a practical implementation of functions like **Filter** and **Compress** (e.g., `tcpdump` and `gzip` [23]) is not memory-safe, and thus allows an attacker to violate the security policy of a program. Suppose that the programmer wants to ensure that **Filter** only interacts with communication channels by opening and reading from its designated input at **read** and writing to its designated output at **wr**, and **Compress** only interacts with communication channels by reading from its configuration files at **cnfg**. However, suppose also that the data-processing action **proc** in **Filter** and the compression action **cmp** in **Compress** perform memory-unsafe operations when passed particular inputs. Then an attacker who can control the inputs to **Filter** could craft a malicious input that injects code that opens a communication channel (e.g., a file) and violates the policy.

However, if the programmer correctly rewrites **Filter** for a suitable *privilege-aware systems*, then the rewritten **Filter** will satisfy such a policy even if it executes code injected by an attacker. Consider a privilege-aware system MiniCap, which is a simplification of the Capsicum capability system now included in the “RELEASE” branch of FreeBSD [11][23]. MiniCap maps each executing process to a two-valued flag denoting if the process has high or low privilege. If a process has high privilege **H**, then it can open communication channels, but if it has low privilege **L**, then it can only read and write to its designated input and output channels. A process on MiniCap begins executing with high privilege, but may invoke the MiniCap primitive **dropcap**, which directs MiniCap to give

the process low privilege, and never give the process the high privilege again. A process thus might invoke `dropcap` after executing safe code that requires high privilege, but before executing unsafe code that requires only low privilege.

MiniCap also allows one process to communicate with another process via a *remote procedure call (RPC)*, in which case the called process begins execution with high privilege, independent of the privilege of the caller. The Capsicum capability system uses RPC in this way, while DIFC systems allow a process to call a process with different privileges via an analogous *gate call* [24].

MiniCap is partially depicted in Fig. II(b) as an automaton M that accepts sequences of privilege-instruction pairs and primitives executed by **Filter**. We call M the MiniCap *monitor* of **Filter**. M accepts a trace of privilege-instruction pairs and primitives if when **Filter** executes the sequence of instructions and primitives, MiniCap grants **Filter** the privilege paired with each instruction. The call and return transitions of M are omitted for simplicity; M transitions on an RPC to the high-privilege state H , and returns from an RPC to the calling state.

Filter's policy can be expressed directly in terms of MiniCap's privileges by requiring that the instructions `read` and `cnfg` execute with high privilege, while the instructions `proc` and `cpr` execute with low privilege. The policy is presented as an automaton Pol in Fig. II(c), where each transition is labeled with a privilege-instruction pair (the label * denotes any label that does not appear explicitly on a transition from the same source state). The traces accepted by Pol are the sequences of instruction-privilege pairs that violate the policy.

For **Filter** to satisfy its policy when it is run on MiniCap, it must use the primitives of MiniCap in a way that is only indirectly related to, and significantly more complex than, its desired policy. In particular, **Filter** must (1) invoke `dropcap` after executing `read` but before executing `proc`, (2) call `Compress` via RPC so that `Compress` executes `cnfg` with high privilege, (3) invoke `dropcap` after executing `cnfg` but before executing `cpr`. This rewritten **Filter** is “modular” across calls and returns, in the sense that the rewritten **Filter** and `Compress` invoke primitives independently of the actions of each other. On practical privilege-aware systems, a process that can be called via RPC cannot necessarily trust its caller, and thus cannot trust information passed by its caller. Thus a practical instrumentation must be modular.

The policy-weaving problem for **Filter** is to take F , its policy Pol , and MiniCap monitor M , and instrument **Filter** to use MiniCap's primitives modularly to satisfy Pol .

2.2 Policy-Weaving Filter via Safety Games

Each policy-weaving problem can be reduced to finding a winning strategy to a safety game. A safety game is played by two players, an Attacker and Defender, and is a transition system in which each state belongs to either the Attacker or the Defender. The goal of the Attacker is to drive the state of the game to an accepting state, while the goal of the Defender is to thwart the Attacker. The game is played in turns: when the game enters an Attacker state, the Attacker

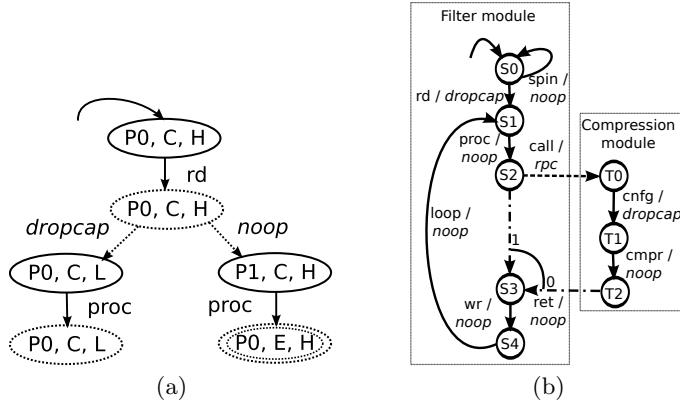


Fig. 2. (a) a selection of transitions of the game G_{ex} that is the product of F , Pol , and M from Fig. 2; (b) a strategy corresponding to a correct instrumentation of **Filter**

chooses the next transition, and when the game enters a Defender state, the Defender chooses the next transition. A strategy for the Defender takes as input a play of the game, and chooses the next transition for the Defender. A winning strategy chooses Defender transitions that never allow the Attacker to drive the state of the game to an accepting state.

From program, policy, and monitor automata, we can construct a game that accepts all policy-violating executions of a version of the program that is instrumented to invoke the primitives of the monitor. The game is constructed by (1) transforming the alphabets of the automata to a common alphabet defined by the instructions, privileges, and primitives, (2) constructing the product of the transformed automata, and (3) transforming the alphabet of the resulting product game so that all Attacker transitions are labeled with program instructions, and all Defender transitions are labeled with system primitives.

A subset of the transitions of the game G_{ex} constructed from F , Pol , and M are shown in Fig. 2(a). Each state of G_{ex} is either an Attacker or Defender state constructed from a triple of a state of F , state of Pol , and state of M , and each state in Fig. 2(a) is labeled with its triple. Each Attacker state and Attacker transition is denoted with a solid circle or line, while each Defender state is denoted with a dotted circle or line. The play “*read, noop, proc*” is accepted by G_{ex} (i.e., is a winning play for the Attacker) because it is an execution in which the instrumented **Filter** does not execute *dropcap* before executing *proc*, causing *proc* to execute with high-privilege, which violates the policy Pol . However, the play “*read, dropcap, proc*” is not accepted by G_{ex} , because it corresponds to an execution in which **Filter** invokes *dropcap*, causing *proc* to execute with low privilege, which satisfies the policy.

One winning Defender strategy to G_{ex} , which corresponds to the correct instrumentation of **Filter** given in §2.1, is presented in Fig. 2(b). The strategy is a transducer that, from its current state, reads an instruction executed by **Filter**, outputs the primitive paired with the instruction on the label of a transition t

(Fig. 2(b) includes a primitive `noop` that denotes that no MiniCap primitive is invoked), transitions on t , and reads the next instruction. The strategy is partitioned into a **Filter** module that chooses what primitives are invoked during an execution of **Filter**, and a **Compress** module that chooses primitives are invoked during the execution of **Compress**. The modules are independent, in that the primitives chosen by the **Compress** module are independent of the instructions and primitives executed by **Filter** before the most recent call of **Compress**.

Solving games constructed from policy-weaving problems efficiently is a hard problem. The game G_{ex} is the product of F , Pol , and M , and thus has a state space whose size is proportional to the product of the sizes of F , Pol , and M (G_{ex} has 128 states). Furthermore, finding modular winning Defender strategies to games is NP-complete in the size of the game. However, in practice, games constructed from policy-weaving problems have a winning strategy whose structure closely matches the structure of one of the input automata. For example, the winning strategy in Fig. 2(b) closely matches the structure of F . Each execution of **Filter** is in state F_n of F when the strategy is in state S_n , and in state C_n of F when the strategy is in state T_n (see Fig. II(a) and Fig. 2(b)). To find winning modular strategies to games efficiently, we apply a novel algorithm that takes a game and an additional, potentially smaller, game called a *scaffold*, and searches for a winning strategy whose structure is similar to that of the scaffold. For G_{ex} , F serves as such a scaffold.

3 Policy Weaving as a Safety Game

3.1 Definition of the Policy-Weaving Problem

The policy-weaving problem is to take a program, a description of a privilege-aware system, and a policy that describes what privileges the program must have as it executes on the system, and to instrument the program so that it always has the privileges required by the policy. We model a program, policy, and privilege-aware system each as a Visibly Pushdown Automaton.

Definition 1. A *deterministic visibly-pushdown automaton* (VPA) for internal actions Σ_I , call actions Σ_C , and return actions Σ_R (alternatively, a $(\Sigma_I, \Sigma_C, \Sigma_R)$ -VPA) is a tuple $V = (Q, \iota, Q_F, \tau_i, \tau_c, \tau_r)$, where: Q is the set of *states*; $\iota \in Q$ is the *initial state*; $Q_F \subseteq Q$ is the set of *accepting states*; $\tau_i : Q \times \Sigma_i \rightarrow Q$ is the *internal transition function*; $\tau_c : Q \times \Sigma_c \rightarrow Q$ is the *call transition function*; $\tau_r : Q \times \Sigma_r \times Q \rightarrow Q$ is the *return transition function*.

For $\widehat{\Sigma} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, each VPA accepts a set of *traces* of (i.e., sequences of actions in) $\widehat{\Sigma}$. Let ϵ denote the empty sequence. Let “.” denote the concatenation of two sequences; for set X , $x \in X$, and $s \in X^*$, $x . s = [x] . s$ and $s . x = s . [x]$, where $[x] \in X^*$ is the sequence containing only x . For sets X_0 and X_1 , let $X_0 . X_1$ be the set of all sequences $x_0 . x_1$ for $x_0 \in X_0$ and $x_1 \in X_1$. Let $\tau : Q^* \times \widehat{\Sigma} \rightarrow Q^*$

map a sequence of states $s \in Q$ (i.e., a *stack*) and action $a \in \widehat{\Sigma}$ to the stack to which V transitions from s on a :

$$\begin{aligned}\tau(q . s, a) &= \tau_I(q, a) . s && \text{for } a \in \Sigma_I \\ \tau(q . s, a) &= (\tau_C(q, a) . q . s) && \text{for } a \in \Sigma_C \\ \tau((q_0 . q_1 . s'), a) &= \tau_R(q_0, a, q_1) . s' && \text{for } a \in \Sigma_R\end{aligned}$$

Let $\rho : \widehat{\Sigma}^* \rightarrow Q^*$ map each trace to the stack that V is in after reading the trace. Formally, $\rho(\epsilon) = \iota$, and for $a \in \widehat{\Sigma}$ and $s \in \widehat{\Sigma}^*$, $\rho(s . a) = \tau(\rho(s), a)$. A trace $t \in \widehat{\Sigma}^*$ is accepted by V if $\rho(t) = q . s$ with $q \in Q_F$. In a trace t , an instance c of a call action is *matched* by an instance r of a return action if c is before r in t , and each instance c' of a call action in t between c and r is matched by a return action r' between c and r . A trace is *matched* if all call and return actions in the string are matched. Let $\mathcal{L}(V)$ be the set of all traces accepted by V . \square

A program is a language of traces of intraprocedural instructions, calls, and returns of the program (e.g., for `Filter` in §2, `spin`, `read`, etc.). Let $\mathbf{Instrs} = (\Sigma_I, \Sigma_C, \Sigma_R)$, let $\widehat{\mathbf{Instrs}} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, and let a program P be an \mathbf{Instrs} -VPA.

A program policy is a language of traces of program instructions paired with privileges. A program's privilege is a system-specific ability (e.g., for MiniCap in §2, a program may have either the high privilege H or the low privilege L). Let \mathbf{Privs} be a set of privileges, and let the set of *privileged executions* of P be $(\widehat{\mathbf{Instrs}} \times \mathbf{Privs})^*$. Let an $(\mathbf{Instrs}, \mathbf{Privs})$ -policy for P be a $(\Sigma_I \times \mathbf{Privs}, \Sigma_C \times \mathbf{Privs}, \Sigma_R \times \mathbf{Privs})$ -VPA (e.g., Fig. II(c)) that accepts all privileged executions that constitute violations.

A privilege-aware monitor is a language of privileged executions interleaved with primitives. The primitives of a privilege-aware system are the set of security-specific system calls that the application can invoke to manage its privileges (e.g., for MiniCap, the system call `dropcap`). Let \mathbf{Prims} be a set of primitives and let the *instrumented executions* of P be $(\mathbf{Prims} . \widehat{\mathbf{Instrs}})^*$. A privilege-aware monitor of P reads an instrumented execution of P , and decides what privilege P has as it executes each instruction. Let an $(\mathbf{Instrs}, \mathbf{Privs}, \mathbf{Prims})$ -privilege-aware monitor M be a $((\Sigma_I \times \mathbf{Privs}) \cup \mathbf{Prims}, \Sigma_C \times \mathbf{Privs}, \Sigma_R \times \mathbf{Privs})$ -VPA.

Definition 2. (Policy-Weaving Problem) Let P be a program with internal, call, and return alphabets $\mathbf{Instrs} = (\Sigma_I, \Sigma_C, \Sigma_R)$. For privileges \mathbf{Privs} , let \mathbf{Pol} be an $(\mathbf{Instrs}, \mathbf{Privs})$ -policy of P . For primitives \mathbf{Prims} , let M be an $(\mathbf{Instrs}, \mathbf{Privs}, \mathbf{Prims})$ -privilege-aware monitor.

Let an *instrumentation function* be a function $I : \widehat{\mathbf{Instrs}}^* \rightarrow \mathbf{Prims}$, and let $I_{tr} : \widehat{\mathbf{Instrs}}^* \rightarrow (\mathbf{Prims} . \widehat{\mathbf{Instrs}})^*$ map each sequence of instructions to the instrumentation of the sequence defined by I : $I_{tr}(\epsilon) = I(\epsilon)$, and for $s \in \widehat{\mathbf{Instrs}}^*$ and $a \in \widehat{\mathbf{Instrs}}$, $I_{tr}(s . a) = I_{tr}(s) . I(s . a) . a$. Let $\mathbf{PrivEx}_M : (\mathbf{Prims} . \widehat{\mathbf{Instrs}})^* \rightarrow (\widehat{\mathbf{Instrs}} \times \mathbf{Privs})^*$ map each instrumented execution to the privileged execution that it induces on M : for primitives p_j , instructions i_j , and privileges r_j , $\mathbf{PrivEx}_M([p_0, i_0, \dots, p_n, i_n]) = [(p_0, r_0), \dots, (p_n, r_n)]$ if and only if $[p_0, (i_0, r_0), \dots, p_n, (i_n, r_n)] \in \mathcal{L}(M)$.

The *policy-weaving problem* $\text{POLWEAVE}(\mathbf{P}, \mathbf{Pol}, \mathbf{M})$ is to find an instrumentation function I such that:

1. I instruments \mathbf{P} to never violate \mathbf{Pol} : $\text{PrivEx}_\mathbf{M}(I_{\text{tr}}(\mathcal{L}(\mathbf{P}))) \cap \mathcal{L}(\mathbf{Pol}) = \emptyset$.
2. I chooses primitives independently of the execution before the most recent call; i.e., I is *modular*. Let $p^0, p^1 \in \text{Img}(I_{\text{tr}})$, (where for a relation R , $\text{Img}(R)$ is the image of R), and $p^0 = p_0^0 \cdot c \cdot p_1^0 \cdot r_0 \cdot p_2^0$, $p^1 = p_0^1 \cdot c \cdot p_1^1 \cdot r_1 \cdot p_2^1$, where call action c is matched by r_0 in p^0 , and is matched by r_1 in p^1 . Let $p_1^0 = a_0, b_0^0, a_1, b_1^0, \dots, a_n, b_n^0$, and let $p_1^1 = a_0, b_0^1, a_1, b_1^1, \dots, a_n, b_n^1$. Then $b_i^0 = b_i^1$ for each i in each such p^0 and p^1 . \square

Defn. 2 formalizes the informal policy-weaving problem illustrated in §2. As discussed in §11, if a policy-weaving problem has a solution I , then it has a solution I^* that may be represented as a VPA transducer T (i.e., a VPA where each action is labeled with input and output symbols). The problem of rewriting program \mathbf{P} to satisfy the policy thus amounts to applying T to \mathbf{P} , using a standard product construction from automata theory.

Privilege-aware systems are typically applied to monitor programs that could run injected code, yet an instrumentation function is defined in Defn. 2 to choose a primitive after each instruction executed by the program. However, this is not a fundamental limitation, as if a programmer or static analysis tool determines that injected code might be run at a particular point in the program, then we can define the monitor so that no primitive other than a `noop` can be invoked by the instrumentation. Conversely, it is not too restrictive to allow an instrumentation function to invoke only a single primitive after each instruction, as we can rewrite the program to execute a sequence of security-irrelevant instructions between which the instrumentation can invoke a sequence of primitives. In [14] App. A, we describe two different privilege-aware systems as VPA.

3.2 From Policy Weaving to Safety Games

Each policy-weaving problem $\text{POLWEAVE}(\mathbf{P}, \mathbf{Pol}, \mathbf{M})$ can be reduced to a *single-entry VPA* (SEVPA [3]) *safety game* that accepts plays corresponding to instrumented executions of \mathbf{P} that violate \mathbf{Pol} when run on \mathbf{M} . A SEVPA safety game is a VPA structured as a set of modules with unique entry points whose transitions are decided in turn by an Attacker and a Defender. The states of the game are partitioned into modules, where the system transitions to a unique module on each call transition.

Definition 3. A SEVPA *safety game* for Attacker internal actions $\Sigma_{I,A}$, Defender internal actions Σ_D , call actions Σ_C , and return actions Σ_R is a tuple $\mathsf{G} = (Q_A, Q_D, Q_0, \iota_0, \{(Q_c, \iota_c)\}_{c \in \Sigma_C}, Q_F, \tau_{I,A}, \tau_D, \tau_R)$, where

- $Q_A \subseteq Q$ is a finite set of *Attacker states*.
- $Q_D \subseteq Q$ is a finite set of *Defender states*. Q_A and Q_D partition the states of the game Q .
- Q_0 is the *initial module*.

- $\iota_0 \in Q_0 \cap Q_D$ is the *initial state*.
- For $c \in \Sigma_C$, Q_c is the *module of c*. The sets $\{Q_c\}_{c \in \Sigma_C}$ and Q_0 are pairwise disjoint, and partition Q .
- For $c \in \Sigma_C$, $\iota_c \in Q_c \cap Q_D$ is the *initial state of c*.
- $Q_F \subseteq Q_0 \cap Q_D$ is the set of *accepting states*.
- $\tau_{I,A} : Q_A \times \Sigma_{I,A} \rightarrow Q_D$ is the *Attacker internal transition function*.
- $\tau_D : Q_D \times \Sigma_D \rightarrow Q_A$ is the *Defender internal transition function*.
- $\tau_R : Q_A \times \Sigma_R \times (Q_A \times \Sigma_C) \rightarrow Q_D$ is the *return transition function*.

The modules are closed under internal transitions: for $x \in \{0\} \cup \Sigma_C$, $q \in Q_x$, and $a \in \Sigma_{I,A}$, $\tau_{I,A}(q, a) \in Q_x$, and for $a \in \Sigma_D$, $\tau_D(q, a) \in Q_x$. A SEVPA safety game is not defined by using an explicit call transition function, because each call on an action c pushes on the stack the calling Attacker state and calling action (we thus call $\Gamma = Q_A \times \Sigma_C$ the *stack symbols of the game*), and transitions to ι_c . The modules of a SEVPA safety game are closed under matching calls and returns: for $x \in \{0\} \cup \Sigma_C$, $c \in \Sigma_C$, $q_x \in Q_x$, $q_c \in Q_c$, and $r \in \Sigma_R$, $\tau_R(q_c, r, (q_x, c)) \in Q_x$.

The *plays* of a SEVPA are defined analogously to the traces of a VPA. Let the *configurations* of G be $C = Q \times \Gamma^*$, let the *attacker configurations* be $C_A = C \cap (Q_A \times \Gamma^*)$, and let the *defender configurations* be $C_D = C \cap (Q_D \times \Gamma^*)$. Let the *Attacker actions* be $\Sigma_A = \Sigma_{I,A} \cup \Sigma_C \cup \Sigma_R$. $\tau_A : C_A \times \Sigma_A \rightarrow C_D$ maps each Attacker configuration and Attacker action to a Defender configuration:

$$\begin{aligned}\tau_A((q, s), a) &= (\tau_{I,A}(q, a), s) && \text{for } a \in \Sigma_{I,A} \\ \tau_A((q, s), a) &= (\iota_c, (q, a) . s) && \text{for } a \in \Sigma_C \\ \tau_A((q, s_0 . s'), a) &= (\tau_R(q, a, s_0), s') && \text{for } a \in \Sigma_R\end{aligned}$$

Because each transition on a Defender action is to an Attacker state and each transition on an Attacker action is to a Defender state, all plays that transition to a defined configuration are in $(\Sigma_D . \Sigma_A)^*$. Let $\rho : (\Sigma_D . \Sigma_A)^* \rightarrow C_D$ map each play of alternating Defender and Attacker actions to the Defender configuration that the game transitions to from reading the play: $\rho(\epsilon) = (\iota_0, \epsilon)$, and $\rho(p . a . b) = \tau_A(\tau_D(\rho(p), a), b)$. A play $p \in (\Sigma_D . \Sigma_A)^*$ is accepted by G if $\rho(p) = (q, \epsilon)$ with $q \in Q_F$. Let $\mathcal{L}(G)$ be the set of all plays accepted by G . \square

Because all accepting states of a game are in the initial module, a game can only accept matched plays. Superscripts denote the VPA or SEVPA game to which various components belong; e.g., Q^G are the states of SEVPA game G .

A *Defender strategy* of a two-player safety game G is a function $\sigma : (\Sigma_A^G)^* \rightarrow \Sigma_D^G$ that takes as input a sequence of Attacker actions, and outputs a Defender action. σ is a *winning strategy* if as long as the Defender uses it to choose his next transition of the game, the resulting play is not accepted by G : formally, $\sigma_{\text{tr}}((\Sigma_A^G)^*) \cap \mathcal{L}(G) = \emptyset$ (for σ_{tr} as defined in Defn. 2). Let σ be modular if it satisfies the condition analogous to a modular instrumentation function (Defn. 2).

Theorem 1. *For each policy-weaving problem $\mathcal{P} = \text{POLWEAVE}(\mathbf{P}, \text{Pol}, \mathbf{M})$, there is a SEVPA safety game $G = \text{PolWeaveGame}(\mathbf{P}, \text{Pol}, \mathbf{M})$ such that each instrumentation function that satisfies \mathcal{P} defines a winning modular Defender strategy of G .*

\mathcal{G} , and each winning modular Defender strategy of \mathcal{G} defines a satisfying instrumentation function of \mathcal{P} .

The intuition behind the construction of \mathcal{G} from $\mathcal{P} = \text{POLWEAVE}(\mathsf{P}, \text{Pol}, \mathsf{M})$ is given in §2.2. From P , we construct a game G_P that accepts all instrumented privileged executions of P . From Pol , we constructed a game G_{Pol} that accepts all instrumented privileged executions that violate Pol . We construct \mathcal{G} as the product of G_P , G_{Pol} , and G_M . Proofs of all theorems stated in §3 and §4 are in [14] App. B.

4 Solving SEVPA Safety Games with Scaffolds

In this section, we present an algorithm **ScafAlgo** that finds a winning modular Defender strategy for a given SEVPA safety game. The algorithm uses an additional, potentially smaller game, which we call a *scaffold*. We present **ScafAlgo** as a non-deterministic algorithm, and demonstrate that a symbolic implementation builds a formula whose size is decided entirely by the size of the scaffold and an additional, tunable independent parameter. We describe a known algorithm for finding modular strategies [4] and a known symbolic algorithm for finding strategies of bounded size [19] as instances of **ScafAlgo**.

4.1 Definition and Key Properties of Scaffolds

The key characteristic of our algorithm is that it finds a winning Defender strategy to a given game using an additional game, called a scaffold, and a specified relation between the states of the scaffold and the states of the game.

Definition 4. (Scaffolds) Let S and G be two SEVPA safety games defined for Attacker actions $\Sigma_{I,A}$, Defender actions Σ_D , call actions Σ_C , and return actions Σ_R . S is a *scaffold* of G under $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$ if and only if:

1. If $q_S \in Q_F^{\mathsf{S}}$ and for $q_G \in Q^{\mathsf{G}}$, $\mathcal{R}(q_S, q_G)$, then $q_G \in Q_F^{\mathsf{G}}$.
2. For $c \in \Sigma_C$, $\mathcal{R}(\iota_c^{\mathsf{S}}, \iota_c^{\mathsf{G}})$.
3. For $a \in \Sigma_{I,A}$, $q_S \in Q_A^{\mathsf{S}}$, and $q_G \in Q_A^{\mathsf{G}}$, if $\mathcal{R}(q_S, q_G)$, then $\mathcal{R}(\tau_{I,A}(q_S, a), \tau_{I,A}(q_G, a))$.
4. For $a \in \Sigma_D$, $q_S \in Q_D^{\mathsf{S}}$, and $q_G \in Q_D^{\mathsf{G}}$, if $\mathcal{R}(q_S, q_G)$, then $\mathcal{R}(\tau_D(q_S, a), \tau_D(q_G, a))$.
5. For $c \in \Sigma_C$, $q_c^{\mathsf{S}} \in Q_c^{\mathsf{S}}$, $q_c^{\mathsf{G}} \in Q_c^{\mathsf{G}}$, $q^{\mathsf{S}} \in Q^{\mathsf{S}}$, $q^{\mathsf{G}} \in Q^{\mathsf{G}}$, if $\mathcal{R}(q_c^{\mathsf{S}}, q_c^{\mathsf{G}})$ and $\mathcal{R}(q^{\mathsf{S}}, q^{\mathsf{G}})$, then $\mathcal{R}(\tau_R(q^{\mathsf{S}}, r, (q_c^{\mathsf{S}}, c)), \tau_R(q^{\mathsf{G}}, r, (q_c^{\mathsf{G}}, c)))$. \square

If so, then \mathcal{R} is a *scaffold relation* from S to G .

Each scaffold relation defines an Attacker bisimulation, with respect to actions, from configurations of the scaffold to configurations of the game. However, the bisimulation over configurations need not relate every accepting configuration of the game to an accepting configuration of the scaffold.

Scaffold relations and modular strategies are connected by the following key property, which provides the foundation for our algorithm. First, we define an

(S, \mathcal{R}, k) -strategy of a game G , which intuitively is a strategy whose structure tightly corresponds to a scaffold S , according to a relation \mathcal{R} from the states of S to those of G . For a game G and $Q' \subseteq Q^G$ such that $\{\iota_c\}_{c \in \Sigma_C^G} \subseteq Q' \subseteq Q^G$, let the *subgame* $G|_{Q'}$ be the game constructed by restricting the states and transition functions of G to the states in Q' . Each subgame G' of G defines a strategy $\sigma_{G'}$ as a VPA transducer. To compute $\sigma_{G'}(a_0, a_1, \dots, a_n)$, $\sigma_{G'}$ uses a_0, a_1, \dots, a_n as the Attacker actions for a play of G' . If G' is in an attacker state p , then $\sigma_{G'}$ transitions on the next unread a_i to $\tau_{I,A}(p, a_i)$. If G' is in a Defender state q , then $\sigma_{G'}$ picks the least Defender action d , under a fixed total ordering of Σ_D , such that $\tau_D(q, d) \in Q'$, outputs d , and transitions to $\tau_D(q, d)$. $\sigma_{G'}$ outputs the Defender action chosen by G' after it reads all of a_0, a_1, \dots, a_n .

Definition 5. For sets A and B , let a relation $\mathcal{R} \subseteq A \times B$ be *k-image-bounded* if for each $a \in A$, $|\{b \mid b \in B, \mathcal{R}(a, b)\}| \leq k$. Let G be a game, let S be a scaffold of G under $\mathcal{R} \subseteq Q^S \times Q^G$, and let $k \in \mathbb{N}$. An (S, \mathcal{R}, k) -Defender strategy σ' of G is a Defender strategy such that for some $\mathcal{R}' \subseteq \mathcal{R}$, $\mathcal{R}' \cap (Q_A^S \times Q_A^G)$ is *k-image-bounded*, $G' = G|_{\text{Img}(\mathcal{R}')}$, and $\sigma' = \sigma_{G'}$. \square

Let game G have a winning Defender strategy, and let S be a scaffold of G under a scaffold relation \mathcal{R} . Then S is a scaffold of some subgame of G' that defines a winning strategy of G , under a finer scaffold relation than \mathcal{R} .

Theorem 2. *Let G have a winning modular Defender strategy, and let S be a scaffold of G under $\mathcal{R} \subseteq Q^S \times Q^G$. Then for some k , there is a winning modular (S, \mathcal{R}, k) -Defender strategy of G .*

4.2 An Algorithm Parametrized on Scaffolds

To find modular winning Defender strategies to games, we can apply Thm. 2 to search for (S, \mathcal{R}, k) -strategies. The algorithm `ScafAlgo`, given in Alg. 1, takes a game G , scaffold S , relation $\mathcal{R} \subseteq Q^S \times Q^G$, and parameter k , and searches for an (S, \mathcal{R}, k) -strategy by searching for an $\mathcal{R}' \subseteq \mathcal{R}$ that satisfies the condition given in Defn. 5.

`ScafAlgo` searches for such an \mathcal{R}' in three main steps. In the first step, `ScafAlgo` non-deterministically chooses a *k*-image-bounded subrelation of \mathcal{R} from the Attacker states of S to the Attacker states of G . Specifically, on line ①, `ScafAlgo` defines such a relation $\mathcal{R}_A \subseteq Q_A^S \times Q_A^G$ by calling a function `nd-bounded-subrel` : $(Q_A^S \times Q_A^G) \times \mathbb{N} \rightarrow (Q_A^S \times Q_A^G)$, where `nd-bounded-subrel`($\mathcal{R} \cap (Q_A^S \times Q_A^G)$, k) is a *k*-image-bounded subrelation of $Q_A^S \times Q_A^G$.

In the second step (lines ②–⑤), `ScafAlgo` constructs a relation $\mathcal{R}_D \subseteq Q_D^S \times Q_D^G$ such that if there is any $\mathcal{R}^* \subseteq Q_D^S \times Q_D^G$ such that $G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}^*)}$ defines a winning strategy of G , then the *candidate strategy* defined by $G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ is a winning strategy of G . On line ②, `ScafAlgo` defines $\mathcal{R}_{D,i} \subseteq Q_D^S \times Q_D^G$ that relates each module-initial state of S to its corresponding module-initial state in G . On line ③, `ScafAlgo` defines $\mathcal{R}_{D,i} \subseteq Q_D^S \times Q_D^G$ that, for each $(p_A, q_A) \in \mathcal{R}_A$ and internal Attacker action $a \in \Sigma_{I,A}^G$, relates the a -successor of p_A to the a -successor of q_A . On line ④, `ScafAlgo` defines $\mathcal{R}_{D,r} \subseteq Q_D^S \times Q_D^G$ that, for each $(p_A, q_A), (s_A, t_A) \in$

Input: G : a VPA safety game.
 S : a scaffold of G
 $\mathcal{R} \subseteq Q^S \times Q^G$: a scaffold relation.

Output: If G has a winning (S, \mathcal{R}, k) -strategy, then it returns a winning (S, \mathcal{R}, k) -strategy. Otherwise, it returns \perp .

```

/* Choose  $\mathcal{R}_A$ : a  $k$ -image-bounded subrelation of  $\mathcal{R}$  that defines
   Attacker states of a candidate strategy. */
```

- 1 $\mathcal{R}_A := \text{nd-bounded-subrel}(\mathcal{R} \cap (Q_A^S \times Q_A^G), k);$
- /* Construct \mathcal{R}_D : a relation to Defender states of the candidate
 strategy defined by \mathcal{R}_A . */
- 2 $\mathcal{R}_{D,\iota} := \{(\iota_c^S, \iota_c^G) \mid c \in \Sigma_G^G\} ;$
- 3 $\mathcal{R}_{D,i} := \{(\tau_{I,A}^S(p_A, a), \tau_{I,A}^G(q_A, a)) \mid (p_A, q_A) \in \mathcal{R}_A, a \in \Sigma_{I,A}^G\} ;$
- 4 $\mathcal{R}_{D,r} := \{(\tau_R^S(p_A, a, s_A), \tau_R^G(q_A, a, t_A)) \mid (p_A, q_A), (s_A, t_A) \in \mathcal{R}_A, a \in \Sigma_R^G\} ;$
- 5 $\mathcal{R}_D := \mathcal{R}_{D,\iota} \cup \mathcal{R}_{D,i} \cup \mathcal{R}_{D,r} ;$
- /* Check if the candidate strategy defined by \mathcal{R}_A and \mathcal{R}_D is a
 winning strategy. */
- 6 $\text{StrWins} := \forall (p_D, q_D) \in \mathcal{R}_D : q_D \notin Q_F \wedge \exists a \in \Sigma_D : (\tau_D^S(p_D, a), \tau_D^G(q_D, a)) \in \mathcal{R}_A ;$
- 7 if StrWins then return $\sigma_{G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}}$ else return \perp

Algorithm 1: ScafAlgo: non-deterministic algorithm that takes a game G , scaffold S , and relation $\mathcal{R} \subseteq Q^S \times Q^G$, and finds a winning modular Defender (S, \mathcal{R}, k) -strategy of G

\mathcal{R}_A and return action $a \in \Sigma_R^G$, relates the r -successor of (p_A, s_A) to the r -successor of (q_A, t_A) . On line [5], ScafAlgo defines $\mathcal{R}_D \subseteq Q_D^S \times Q_D^G$ as the union of \mathcal{R}_ι , $\mathcal{R}_{D,i}$, and $\mathcal{R}_{D,r}$.

In the third step (lines [6] and [7]), ScafAlgo checks if the candidate strategy defined by $G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ is a winning strategy of G . On line [6], ScafAlgo defines $\text{StrWins} : \mathbb{B}$, which is true if and only if for each Defender-state of the candidate strategy, the state is not an accepting state of the game, and there is some action that the Defender can take to reach some Attacker-state of the candidate strategy. On line [7], ScafAlgo returns the strategy defined by $G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ if and only if $G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ is a winning strategy. Otherwise, ScafAlgo returns failure.

Theorem 3. Let G be a game, let S be a scaffold of G under $\mathcal{R} \subseteq Q^S \times Q^G$, and let k be a positive integer. If $\sigma = \text{ScafAlgo}(G, S, \mathcal{R}, k)$, then σ is a winning Defender strategy for G . If G has a winning Defender strategy, then for each scaffold S and scaffolding relation $\mathcal{R} \subseteq Q^S \times Q^G$, there is some k such that $\text{ScafAlgo}(G, S, \mathcal{R}, k)$ is a winning Defender strategy of G .

A deterministic implementation of ScafAlgo runs in worst-case time exponential in the number of Attacker states. However, a symbolic implementation of ScafAlgo can represent its input problem with a formula whose size depends only on the scaffold, and the tunable parameter k . Assume that each component of an input game G is given as interpreted symbolic functions and predicates (i.e., states and actions are given as domains, and the transition functions are given

as interpreted functions), and that the relation \mathcal{R} is given as an interpreted relation. Then **ScafAlgo** can be implemented by reinterpreting its steps to build a symbolic formula **StrWins** (line 6) whose models correspond to values of \mathcal{R}_A and \mathcal{R}_D for which $G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ defines a winning strategy.

The size (i.e., the number of literals in) of **StrWins** is determined by S and k . The universal quantification on line 6 is bounded, and can thus be encoded as a finite conjunction; the nested existential quantification can then be Skolemized. To check the membership $(\tau_D^S(p_D, a), \tau_D^G(q_D, a)) \in \mathcal{R}_A$, we can apply the fact that \mathcal{R}_A is a k -bounded-image relation to represent the membership check with k disjuncts. From these observations, the size of the **StrWins** formula can be bounded by $O(|Q_A^S|^2 k^3)$.

Two known algorithms for finding modular strategies can be defined as **ScafAlgo** applied to degenerate scaffolds. A naive implementation of the original algorithm presented for finding modular strategies [4] can be defined as **ScafAlgo** applied to the game itself as a scaffold. A symbolic algorithm for finding strategies of bounded size [19], generalized to VPA games, can be defined as **ScafAlgo** applied to a scaffold with a single Attacker and Defender state for each module. The known algorithms are thus **ScafAlgo** applied to scaffolds that have complete information and no information about their games, respectively. However, any game defined as a product of “factor” games has as a scaffold the product of any subset of its factors. In particular, for each policy-weaving game, we can automatically construct scaffolds from products of any subset of the program, monitor, and policy automata.

5 Experiments

In this section, we discuss experiments that evaluate the reduction from policy-weaving problems to safety games presented in §3, and the scaffold-based game-solving algorithm presented in §4. The experiments were designed to answer two questions. First, by reducing policy-weaving problems to solving games, can we efficiently instrument practical programs for a real privilege-aware system so that they satisfy practical high-level policies? Second, which scaffolds allow our scaffolding game-solving algorithm to most efficiently solve games constructed by our policy-weaving algorithm?

To answer these questions, we instantiated our policy-weaving algorithm to a policy weaver for the Capsicum [23] capability operating system. We collected a set of six UNIX utilities, given in Tab. I, that have exhibited critical security vulnerabilities [16, 21, 22, 23]. For each utility, we defined a policy that describes the capabilities that the program must have as it executes. The policies were defined by working with the Capsicum developers, or using general knowledge of the utility. Detailed descriptions of the policies for each utility are given in [13].

We applied our Capsicum policy-weaver to each utility and its policy, with each scaffold defined as a product of some subset of the program, policy, and monitor. The data from all scaffolds is given in [14] App. D; Tab. II presents data

Table 1. Performance of the Capsicum policy weaver. Column “LoC” contains lines of C source code (not including blank lines and comments) and “Pol. States” contains the number of states in the policy. For the trivial scaffold “Triv.”, intermediate scaffold “Prog.-Pol.”, and complete scaffold “Prog.-Pol.-Mon.”, k contains the simulation bound, “Times” contains the times used to find a strategy, and “Prims.” contains the number of callsites to primitives inserted. “-” denotes a time-out of 20 minutes.

Name	LoC	Pol. States	Scaffolds								
			Triv.			Prog.-Pol.			Prog.-Pol.-Mon.		
			k	Time	Prims.	k	Time	Prims.	k	Time	Prims.
bzip2-1.0.6	8,399	12	12	-	-	1	0:04	6	1	0:09	6
fetchmail-6.3.19	49,370	12	7	-	-	1	1:13	5	1	1:39	5
gzip-1.2.4	9,076	9	12	-	-	1	1:47	15	1	-	-
tar-1.25	108,723	12	3	3:47	15	1	1:20	15	1	-	-
tcpdump-4.1.1	87,593	12	15	-	-	1	0:30	6	1	0:45	6
wget-1.12	64,443	21	7	0:43	11	1	0:25	11	1	18:59	11

for several illustrative scaffolds: the trivial scaffold “Triv.” defined in §4.2, the product of the program and policy “Prog.-Pol”, and the product of all program, policy, and monitor “Prog.-Pol.-Mon.” For each scaffold, we measured how long it took our weaver to find a strategy, and with what minimum simulation bound (i.e., value of k from §10) it either found a strategy or timed out. The results for each scaffold are in the subcolumns of “Scaffolds” in Tab. I, with each simulation bound in subcolumn “ k ,” and each time in subcolumn “Time.”

The results indicate that while many scaffolds give similar results for some practical problems, an *intermediate* scaffold constructed as a product of some but not all of the inputs, e.g. Prog.-Pol., leads to the best performance. The difference in performance could be due to the fact that a scaffold with little information about the structure of its game (e.g., “Triv.”) generates a formula that allows many transitions between a small set of states in a candidate strategy, while a scaffold with total information (e.g., Prog.-Pol.-Mon.) generates a formula that allows few transitions between a large set of states in a candidate strategy. An intermediate scaffold strikes a balance between the two, generating a formula that allows a moderate number of transitions between a moderate set of states. The time taken to find a strategy does not directly depend on the size of the original program, because we apply several optimizations when constructing a policy-weaving game that cause the size of the constructed game to depend only on the size of program modules relevant to a given policy.

For each scaffold, the column “Prims.” contains the number of callsites to primitives dictated by the strategy. Our current game-solving algorithm does not minimize the number of such callsites, and as a result, the number of callsites may be larger than necessary. Moreover, in the current implementation, the number of callsites does not depend on the scaffold used to find a strategy for the game.

6 Related Work

Privilege-aware operating systems: Decentralized Information Flow Control (*DIFC*) operating systems such as Asbestos [9], HiStar [24], and Flume [18] manage privileges describing how information may flow through a system, and provide primitives that allow an application to direct flows by managing the labels of each object in the system. Tagged memory systems such as Wedge [6] enforce similar policies per byte of memory by providing primitives for managing memory tags. Capability operating systems such as Capsicum [23] track the capabilities of each process, where a capability is a file descriptor paired with an access right, and provide primitives that allow an application to manage its capabilities.

Our work complements privilege-aware operating systems by allowing a programmer to give an explicit, high-level policy, and automatically rewriting the program to satisfy the policy when run on the system. Prior work in aiding programming for systems with security primitives automatically verifies that a program instrumented to use the Flume primitives enforces a high-level policy [15], automatically instruments programs to use the primitives of the HiStar to satisfy a policy [8], and automatically instruments programs [12] to use the primitives of the Flume OS. However, the languages of policies used in the approaches presented in [8][12] are not temporal and cannot clearly be applied to other systems with security primitives, and the proofs of the correctness of the instrumentation algorithms are ad hoc. The work in [13] describes the approach in this paper instantiated to a policy weaver for Capsicum. This paper describes how the work in [13] may be generalized to arbitrary privilege aware systems, and describes the novel game-solving algorithm applied in [13].

Inlined Reference Monitors: An Inlined Reference Monitor (*IRM*) [110] is code that executes in the same memory space as a program, observes the security-sensitive events of the program, and halts the program immediately before it violates a policy. IRMs have shortcomings that prohibit them from monitoring many practical programs and policies. Because an IRM executes in the same process space as the program it monitors, it cannot enforce policies throughout the system. Furthermore, an IRM must be able to monitor security-sensitive events of a program throughout the program’s execution, but there are known techniques to subvert an IRM [1]. Privilege-aware operating systems address the shortcomings of IRM by monitoring policies in the operating system, and providing a set of primitives that an application invokes to direct the operating system. The primitives are distinct from the security-sensitive events of interest.

Safety Games: Automata-theoretic games formalize problems in synthesizing reactive programs and control mechanisms [2]. Alur et. al. give an algorithm that takes a single-entry recursive state machine and searches for a strategy that is modular, as defined in §3, and show that this problem is NP-complete [4]. Recursive state machines are directly analogous to SEVPA [3]. Madhusudan et. al. give a set of symbolic algorithms that find a winning strategy for a given

game whose transition relation is represented symbolically [19]. The practical contribution of our work is that we express the emerging and practical problem of rewriting programs for privilege-aware operating systems in terms of such games. We also give an algorithm for finding modular strategies that can be instantiated to a symbolic implementation of the algorithm of [4], to the “bounded-witness” algorithm of [19].

Jobstmann et al. [17] consider the problem of rewriting a program to satisfy a Linear Temporal Logic (LTL) specification, and reduce the problem to an LTL game. Their reduction constructs Defender actions (i.e., “system choices” [17]) from failure-inducing assignments of expressions in the program, whereas our work constructs Defender actions from a set of security system calls. Also, they reduce program repair to finite-state games, while our reduction relies crucially on modular strategies for SEVPA games. Thus, while the work of Jobstmann et al., like ours, is formalized in terms of automata games, the approaches differ in both the meaning of actions performed by players of the game, and the fact that we require a context-sensitive model of a target program.

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: CCS (2005)
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. In: FOCS (1997)
3. Alur, R., Kumar, V., Madhusudan, P., Viswanathan, M.: Congruences for Visibly Pushdown Languages. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1102–1114. Springer, Heidelberg (2005)
4. Alur, R., La Torre, S., Madhusudan, P.: Modular Strategies for Recursive Game Graphs. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 363–378. Springer, Heidelberg (2003)
5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC (2004)
6. Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: Splitting applications into reduced-privilege compartments. In: NSDI (2008)
7. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (1977)
8. Efstathopoulos, P., Kohler, E.: Manageable fine-grained information flow. In: EuroSys (2008)
9. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R.: Labels and event processes in the Asbestos operating system. In: SOSP (2005)
10. Erlingsson, Ú., Schneider, F.B.: IRM enforcement of Java stack inspection. In: IEEE SP (2000)
11. FreeBSD 9.0-RELEASE announcement (January 2012), <http://www.freebsd.org/releases/9.0R/announce.html>
12. Harris, W.R., Jha, S., Reps, T.: DIFC programs by automatic instrumentation. In: CCS (2010)
13. Harris, W.R., Jha, S., Reps, T.: Programming for a capability system via safety games. Technical Report TR1705, University of Wisconsin-Madison (April 2012)

14. Harris, W.R., Jha, S., Reps, T.W.: Secure programming via visibly pushdown safety games. Technical Report TR1710, University of Wisconsin, Dept. of Computer Sciences (April 2012)
15. Harris, W.R., Kidd, N.A., Chaki, S., Jha, S., Reps, T.: Verifying Information Flow Control over Unbounded Processes. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 773–789. Springer, Heidelberg (2009)
16. Izdebski, M.: bzip2 ‘BZ2_decompress’ function integer overflow vuln. (October 2011)
17. Jobstmann, B., Griesmayer, A., Bloem, R.: Program Repair as a Game. In: Etesami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
18. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: SOSP (2007)
19. Madhusudan, P., Nam, W., Alur, R.: Symbolic computational techniques for solving games. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
20. Naraine, R.: Symantec antivirus worm hole puts millions at risk. eWeek.com (2006)
21. Ubuntu sec. notice USN-709-1 (2009), <http://www.ubuntu.com/usn/usn-709-1/>
22. Vuln. note VU#381508 (July 2011), <http://www.kb.cert.org/vuls/id/381508>
23. Watson, R.N.M., Anderson, J., Laurie, B., Kennaway, K.: Capsicum: Practical capabilities for UNIX. In: USENIX Security (2010)
24. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: OSDI (2006)

Alternate and Learn: Finding Witnesses without Looking All over

Nishant Sinha¹, Nimit Singhania¹, Satish Chandra², and Manu Sridharan²

¹ IBM Research Labs, India

² IBM T. J. Watson Research Center, U.S.A.

Abstract. Most symbolic bug detection techniques perform search over the program control flow graph based on either forward symbolic execution or backward weakest preconditions computation. The complexity of determining inter-procedural all-path feasibility makes it difficult for such analysis to judge up-front whether the behavior of a particular caller or callee procedure is relevant to a given property violation. Consequently, these methods analyze several program fragments irrelevant to the property, often repeatedly, before arriving at a goal location or an entrypoint, thus wasting resources and diminishing their scalability.

This paper presents a systematic and scalable technique for *focused* bug detection which, starting from the goal function, employs alternating backward and forward exploration on the program call graph to lazily infer a small *scope* of program fragments, sufficient to detect the bug or show its absence. The method learns *caller* and *callee* invariants for procedures from failed exploration attempts and uses them to direct future exploration towards a scope pertinent to the violation.

1 Introduction

Even though sophisticated static analysis methods for bug detection exist [6][12][18][16], the scalability of these methods is restricted. This is somewhat surprising given that most bugs can be attributed to program behavior in a small set of program regions, i.e., a *small scope* [16][1].

We believe that the common drawback of these methods is that they cannot *focus* on a small set of pertinent program regions that trigger the bug. Such focusing is not easy: a static analysis tool encounters plenty of code irrelevant to a particular bug, but such code is not obviously irrelevant before it is analyzed. Furthermore, the tool may repeatedly re-analyze such irrelevant code, thus wasting resources without finding a witness.

Consider a few examples illustrating the need of focusing. (a) Suppose a *goal* function with a potential null dereference makes a virtual call with 100 possible targets, none of which are relevant to the bug. Exploring all these targets is wasteful, and therefore it is necessary to restrain the forward search to only a subset of callees. (b) Alternatively, consider a goal function g invoked in a large number of call contexts (exponential in the depth of call graph, in the worst case). If the analysis begins from *main* procedure, it is likely that many irrelevant program fragments will be encountered and analyzed before reaching g . Therefore, a goal-driven backward search is necessary for focusing.

Based on above observations, we may conceive of a potentially effective technique that performs backward expansion from a goal function g in a *small scope* centered around g . Effective discovery of such a scope in practice is non-trivial: previous

work [16] employed a strategy based on breadth-first expansion from the goal function, but this may be inefficient if callers or callees far away from the goal need to be explored.

In this paper, we propose a new focused method to perform inter-procedural analysis for detecting bugs. The strategy performs a systematic search around the goal function g with the aim of either inferring a small scope which can trigger the bug or, in some cases, proving the absence of it. Note that finding a witness path to an error location in g requires finding a feasible *call context* for g . This call context consists not only of a set of transitive (backward) callers of g , but also (forward) callees invoked by g on the path to the error function. Based on this observation, our method *alternates* between forward and backward exploration in the call graph to detect a violation and backtracks whenever it fails to find a feasible call context. During alternation, forward expansion takes priority over backward expansion. This is crucial because forward expansion proves infeasibility of the error at the current caller level, and avoids further backward expansion into irrelevant program fragments, thus discovering small program scopes in practice.

The alternating expansion method, despite being lazy, may revisit several irrelevant program regions (e.g., error-free call contexts), re-analyze them and perform wasteful backtracks. Such unfocused exploration clearly reduces the efficiency of the analyzer. Therefore, to improve focus, we propose to *learn*, on-demand from exploration failures, *caller/callee* invariants that over-approximate the caller/callee data values respectively. These invariants contain specific facts which induced the failure and help avoid similar failures later by not re-exploring irrelevant callers/callees.

The proposed method may be viewed as an instance of the general DPLL paradigm, *explore-fail-learn-backtrack*, applied directly to the program call graph representation instead of operating at a fine-grained inter-procedural control flow graph level [18]. Because there may be large number of call contexts to a particular procedure, the backward search tries to efficiently explore the set of call contexts in a depth-first manner, backtracks from failures, and exploits caller/callee invariants inferred from failures to prune future search. The forward expansion assists the backward search to infer early failures, akin to how theory propagation assists in finding conflicts during DPLL search.

In our preliminary experiments with industrial Java benchmarks, we found that alternating scope expansion is crucial to get some benchmarks to finish in a reasonable time. Learning reduced the number of call graph edges visited, but this reduction is not always able to compensate for the overhead of computing invariants.

The key contributions of the paper are as follows:

- A scalable bug detection method ALTER that performs alternating backward and forward search (Sec. 4) to lazily infer a small scope around the goal function, sufficient to detect a witness. A symbolic intra-procedural *local* summary for each procedure (Sec. 3) forms the basis of efficient inter-procedural alternating expansion.
- A systematic technique to learn a program scope pertinent for bug-detection by inferring caller and callee invariants for procedures from failed explorations (Section 5).
- An experimental evaluation (Sec. 6) that shows the effectiveness of our techniques.

2 Motivating Examples and Overview

2.1 Alternating Scope Expansion

Consider the program App1 in Fig. 1; here, the goal function is `A.init`, where a potential null dereference may occur at line 11 because the class A's local field `this.srcs` (non-null) is shadowed by the local parameter variable `srcs`.

ALTER first computes the local error condition for the goal at line 11 in `A.init`: $\phi := (\text{srcs}_{A.\text{init}} = \text{null})$, where `srcs_{A.\text{init}}` refers to the `srcs` parameter of `A.init` (the extra constraints arising from the conditional at line 8–10 are simplified away). Now, ALTER must examine callers of `A.init`, namely `T.T` and `A.A`. Carrying out a *backward expansion* for `T.T`, ALTER composes the local path condition for calling `A.init` inside `T.T`, with ϕ . This composition yields false, because the `srcs` parameter of `T.T` must be non-null for execution to pass line 16 of `T.T`. Next, ALTER carries out backward expansion to include `A.A`, and another backward expansion to include `M.M`, which is a caller of `A.A`. At this point, it carries out a *forward expansion* to bring `M.makeList` in scope. Now, the side effect summary of `M.makeList` can prove —the return value of `M.makeList` cannot be null— that the call context `M.M → A.A → A.init` cannot lead to error. Thus, ALTER is able to show the absence of null dereference in `A.init` by alternating backward/forward expansion starting from the goal location in `A.init`.

```

1  class A implements C {
2      List srcs;
3      A(List srcs, Rect b) {
4          init(srcs, b);
5      }
6      void init(List srcs, Rect b) {
7          this.srcs = new Vector();
8          if (srcs != null) {
9              this.srcs.addAll(srcs);
10         }
11         if (srcs.size() != 0) {...}
12     }
13 }
14 class T extends A {
15     T(List srcs, Rect b) {
16         if (srcs.isEmpty()) return;
17         init(srcs, b);
18     }
19 }
20
21  class M extends A {
22      M(C src, C alpha) {
23          List srcs; Rect b;
24          srcs = makeList(src, alpha);
25          b = makeBounds(src, alpha);
26          super(srcs, b);
27      }
28      List makeList(C s1, C s2) {
29          List ret = new ArrayList(2);
30          ret.add(s1);
31          ret.add(s2);
32          return ret;
33      }
34      class N {
35          void foo(C src, C alpha) {
36              C m = new M(src, alpha);
37              ...
38          }
39      }
40 }
```

Fig. 1. App1 example, based on a fragment of the batik open-source benchmark

Focused Exploration. Note how ALTER performs a focused search by avoiding exploration of irrelevant program regions which are in the nearby scope, i.e., functions `makeBounds` in `M.M`, `isEmpty` in `T.T`, `addAll` in `A.init`, `add` in `M.makeList` and other callers of `M.M` and `T.T`. See Figure 2. The method names in bold are the only ones visited in this process. In particular, note how forward expansion of `M.makeList` ensures early backtrack and avoids further backward expansion from `M.M`. Without

alternating forward and backward expansion, the analysis would expand backward to callers of `M.M`, such as `N.foo` and its callers in Figure 2, which are irrelevant for the goal.

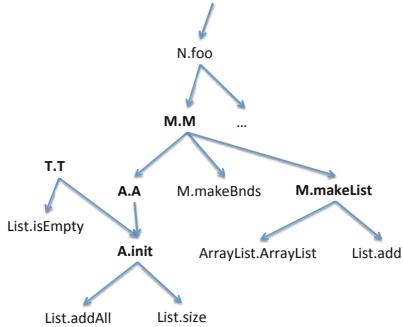


Fig. 2. Call Graph of App1

```

1 class App2 {
2     void runA () {
3         {... foo(new A()); ...}
4     void runB () {
5         {... foo(new B()); ...}
6     //classes A, B extend class C
7     int foo (C c) {
8         if (*) return bar(c,1);
9         else return bar(c,2);
10    }
11    int bar (C c, int i) {
12        return c.compute(i);
13    }
14 }

```

Fig. 3. App2 example

2.2 Learning Pertinent Scopes

In Fig. 3, the function `bar` contains a potential null dereference if the parameter `c` is `null`; `bar` is called by `foo` at two sites, which in turn, is called by `runA` and `runB` with newly allocated objects. Let us denote the local parameter `c` of `foo` by c_{foo} , and of `bar` by c_{bar} . ALTER begins analysis by building the local *error condition* for `bar`, i.e., $\phi := (c_{bar} = \text{null})$, which is satisfiable if the parameter `c` gets the *null* value under some call context to `bar`. To find such a context, ALTER performs backward search in a depth-first manner among callers of `bar`. The two call sites in `foo` for `bar` are analyzed individually; suppose the first call site foo_1 at line 8 is analyzed first. ALTER propagates ϕ backward, resulting in $\phi' := (c_{foo} = \text{null})$. Here, c_{foo} is substituted by the actual called value, which is a heap-allocated object represented as `alloc(A)`; so, $\phi'' = (\text{alloc}(A) = \text{null})$, which is unsatisfiable. Because the current context `runA → foo1 → bar` fails to find a witness, ALTER backtracks and tries the other caller `runB` for `foo`. Again, it fails, and backtracks further to try a different call site for `bar`: (site foo_2 at line 9). ALTER continues to try callers `runA` and `runB` again; however, no witness is found and the search terminates.

Focused Exploration. Note, however, that exploring `runA` and `runB` for the second call site foo_2 to `bar` in `foo` is redundant because we already know from exploring the first call site foo_1 that ($c_{foo} \neq \text{null}$) for all callers to `foo` and hence no witness is possible via the callers of `foo`. A naive exploration technique may therefore explore the same callers redundantly without success because it does not *learn* from failed search attempts. The proposed algorithm therefore incorporates *learning* from failed exploration (Sec. 5): the learned information helps prune away the irrelevant program scope and focus search towards relevant regions.

3 Preliminaries and Intra-procedural Analysis

We refer to the program statement with the violation, e.g., a null dereference, as the *goal location*. Also, the procedure having the goal location is called as the *goal procedure*. We say that a procedure f in an application is an *entrypoint* for the application if f is a public method. An entry point is *relevant* if it may call the goal procedure g transitively. Given a set of relevant entrypoints E , our analysis tries to find a (inter-procedural) feasible path, called *witness*, to the goal location from some entrypoint in E . We refer to such a path as a *global witness*. In contrast, any feasible path which terminates at the goal location but does not begin at a relevant entrypoint is said to be a *local witness*.

For a procedure f , the *input* (*output*) variables consist of the non-local variables and fields read (written) by some statement in f ; the output variables also include two special variables *ret* and *exc* denoting the returned data and exception values from f respectively. A *symbolic state* $s = (\psi, \sigma)$ at a location l is a tuple consisting of a *reachability* predicate ψ and a map σ from scalar variables, fields and arrays to their symbolic values (terms). The predicate ψ represents the condition under which l can be reached via a given set of paths terminating at l . The map σ represents the symbolic values of variables obtained under the same set of paths. Both fields and arrays are modeled as mathematical maps from object references (integers) to their values. We do not distinguish between fields and arrays in our presentation; we use the term *fields* to refer to both. Loops are transformed to tail-recursive functions.

Local Summary for a Procedure. Classical inter-procedural program analysis [20][19] intertwines procedure summary computation with summary composition: the (global) summary G_f for a function f is obtained after composing f 's local behaviors with the summaries of all the callees of f . Such close coupling of summary computation and composition makes it hard to selectively explore the callees for a given goal location in f . For selective exploration, our approach decouples summary computation with composition: we analyze a procedure f in isolation and compute a *local summary* L_f for f *independent* of its callers and callees (referred to as the *environment* of f). The local summary L_f over-approximates the effect of both the callers and the callees of f and has two benefits: (a) we need not re-analyze f for different call contexts, and (b) we can utilize summaries from the environment of f to improve the precision of L_f in a lazy, goal-driven manner, and obtain G_f in the limit. To analyze f independent of its callees, we resort to *structural abstraction* [24][1]: all outputs of each potential callee g of f are modeled using fresh symbolic variables (Skolem constants) denoting arbitrary values that the call to g may return. These Skolem constants (skolems, in short) over-approximate the output values of g and hence allow us to conservatively incorporate g 's behavior in the summary of f .

Formally, the local summary L_f consists of three components: a *side effect* summary, a set of *call site* summaries and a set of *error conditions* (ECs). The *side effect* summary of f is a map from the outputs of f to their symbolic values in terms of inputs of f and captures the data flow from inputs to outputs along all possible paths of f . Let en_f denote the entry location of f . For each call site f_j in f , we compute a *call site* summary at f_j denoted by a symbolic state $s = (\psi, \sigma)$, where ψ denotes the all-path reachability condition of f_j from en_f and the state σ contains the symbolic values of variables and fields obtained along each path to f_j and expressed in terms of inputs of f . Finally, for each goal location l in f , the error condition (EC) predicate ϕ is obtained by conjoining

the all-paths reachability condition from en_f to l with the violation condition, e.g., the null dereference predicate ($v = \text{null}$) for a variable v .

If f has no callees or all the callees are inlined into f , then all the components of L_f are precise, i.e., L_f contains the precise symbolic values of outputs along each path and precise reachability conditions for each error location from en_f . However, if we employ structural abstraction to decouple the callees of f , L_f becomes over-approximate. In particular, an EC ϕ may now contain skolems and satisfiability of ϕ no longer implies that an actual local witness to l exists. Note that ϕ may also contain input variables to f and hence a local witness may not extend to any global witness. Both these sources of imprecision in L_f are removed on-demand during the inter-procedural exploration phase (cf. Sec. 4) for finding a global witness.

Summary Computation. We compute the summary for a function f by a forward all-path analysis algorithm which propagates the symbolic state along all paths of f precisely starting from en_f . We use program expressions to represent symbolic states precisely and propagate states by employing precise transformers for each statement in f (structural abstraction is applied at each call location). To avoid path explosion as well as maintain precision, the algorithm merges symbolic states at join nodes by guarding the incoming symbolic value along each edge by the corresponding path condition and representing the merged state using an *if-then-else* (ite) term compactly. The details of Java statement transformers can be found in [4] and merge operation in [13][21] and are omitted in the interest of space. During propagation, we compute the ECs at each goal location, the call site summaries at each call location and the side effect summary at the exit location of f .

```

int p(int x){
    if(x < 10)
        error();
    return x - 10;
}

int q(int y){
    if(y > 6){
        int z = t(y);
(1)      int a = p(z);
(2)      int b = r(y, z); (3)
        return (a + b);
    }
    return 0;
}

```

```

int r(int u, int v){
    if(u > v)
        return p(u); (1)
    else
        return p(v); (2)
}

int s(int c){
    return r(c, 10); (1)
}

int t(int d){
    return d * 2;
}

```

Fig. 4. Program P

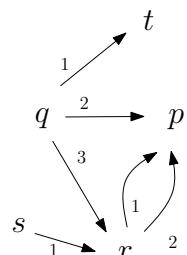


Fig. 5. Call Graph of Program P

Example. Consider program P in Fig. 4. The summary for the return value of r is $\text{ite}((u > v), sk_1^p, sk_2^p)$ where $u(v)$ is the initial value for parameter $u(v)$ in r and $sk_1^p(sk_2^p)$ is the return value of p at call site $r_1(r_2)$. The call site summary for call

site r_1 in \mathbf{r} is (ψ, σ) , where, $\psi := (u > v)$ and $\sigma := [\mathbf{u} \rightarrow u, \mathbf{v} \rightarrow v]$. Then, the error condition ϕ for the violation (function $error()$ in \mathbf{p}) in \mathbf{p} is $\phi := (x < 10)$.

4 Backward, Forward and Alternating Expansions

Recall that an EC ϕ local to f is imprecise because it contains inputs of f and skolems from callees of f , both of which are unconstrained. Hence even if ϕ is satisfiable, neither a local nor a global witness may exist. To search for a global witness, we perform inter-procedural analysis by expanding the scope around the goal function iteratively using a combination of *forward* and *backward* expansion. Forward expansion replaces skolems in ϕ with the actual return values of callees while backward expansion substitutes the inputs with the actual input values for a calling context of f .

Backward. Consider an EC ϕ at entry of a procedure f such that the satisfiability of ϕ implies a local witness to the goal location from f . To propagate back ϕ into a particular caller h of f at site h_k , we use the call site summary (ψ, σ) at h_k . This summary allows us to express the inputs in ϕ directly in terms of inputs of h without re-analyzing h . For every input i in ϕ , let $Val(i, h_k)$ denote the value of i in the symbolic state σ before the call at h_k . Backward propagation is achieved by computing $\phi' := (\phi \wedge CC(h_k))$:

$$CC(h_k) := (\bigwedge_{i \in in_\phi} (i = Val(i, h_k)) \wedge \psi)$$

where $CC(h_k)$ consists of constraints expressing each input i in set of inputs in_ϕ of ϕ , in terms of actual symbolic values at the call site and the all-path reachability condition ψ from entry of h to h_k ¹. The procedure $\text{EXPANDBWD}(h_k \rightarrow f, \phi)$ computes $CC(h_k)$.

Forward. Suppose we want to expand a skolem sk at a call site f_j in f , where sk corresponds to an output variable, say ret , in a callee g . We first obtain the summary expression sum_{ret} for ret from the side-effect summary of g and then substitute the inputs in sum_{ret} with the actual values obtained from the call site summary at f_j . More precisely, the forward expansion constraint for sk is $SC(sk) := SC_1(sk) \wedge SC_2(sk)$. Here, $SC_1(sk)$ contains the summary expression, i.e., $SC_1(sk) := (sk = sum_{ret})$. Note that sum_{ret} depends on the set of inputs In of g and skolems Sk corresponding to callees of g . So, we *raise* the inputs In to the caller by using the call site values $Val(i, f_j)$ (defined above) from call site f_j , i.e., $SC_2(sk) := \bigwedge_{i \in In} (i = Val(i, f_j))$. In sum_{ret} , we also replace each $sk \in Sk$ by a fresh value sk' using a *contextualization* scheme which records the fact that sk' corresponds to the call from f_j to g . The details of the scheme can be found in the full version [23] of this paper and is omitted for clarity. Note that sk' may be expanded forward in a similar way as sk . Let the procedure $\text{EXPANDFWD}(f, \phi)$ compute the skolem constraints SC (recursively, if required) for the set of skolems Sk' in ϕ , i.e., $SC := \bigwedge_{sk \in Sk'} SC(sk)$.

Example. In Fig. 4, the initial EC in \mathbf{p} is $\phi_1 := (x < 10)$. Suppose, we need to propagate EC ϕ_1 back to caller \mathbf{q} at call site q_2 . We start by computing $CC(q_2) := (y > 6 \wedge x = sk^t)$ where sk^t is skolem for call to \mathbf{t} at site q_1 . On *backward* propagation and simplification, EC becomes $\phi_2 := (\phi_1 \wedge CC(q_2)) \equiv (sk^t < 10 \wedge y > 6)$. Now, we

¹ Similar to [4], we also add constraints for handling virtual calls; described in [23].

expand *forward* the skolem sk^t in ϕ_2 using $SC := (sk^t = d * 2) \wedge (d = y)$. Finally, the EC is $\phi_3 := \phi_2 \wedge SC \equiv (y * 2 < 10 \wedge y > 6)$, which is unsatisfiable.

In practice, instead of conjoining constraints, we substitute the actual values for inputs and summaries for skolems in the error condition ϕ . This assists simplification before invoking a constraint solver to check for satisfiability of ϕ . Note that iterative forward or backward expansion may not terminate due to recursive function calls. Therefore, we impose fixed bounds to terminate expansion under recursion. Similarly, we cannot expand a skolem (expand backward) if the source code of the corresponding callee (caller) is not available to the analyzer.

4.1 Alternating Expansion

Alg. 11 describes the alternating expansion algorithm. The main procedure ALTER takes the goal function g and an EC ϕ from summary of g as input and performs a backtracking based search over the program call graph. In a particular iteration with EC ϕ local to function f , ALTER proceeds as follows. First, ALTER expands the skolems in ϕ using EXPANDFWD to obtain the corresponding summary constraints SC . If $\phi \wedge SC$ is satisfiable, ALTER expands all the callers of f ($CALLERS(f)$) using EXPANDBWD in a depth-first manner iteratively. Given a caller h with call site h_k , EXPANDBWD returns the call context constraints $CC(h_k)$ for h_k , which express the inputs of f in terms of inputs of h . ALTER then recursively proceeds to analyze h with the new error condition $\phi' := (\phi \wedge SC \wedge CC(h_k))$ obtained by conjoining both forward and backward expansion constraints with the previous ϕ .

If the EC ϕ at any moment during alternating expansion is infeasible (UNSAT), it indicates an *exploration failure*, i.e., no further backward/forward search will yield a global witness. In this case, ALTER *backtracks* to the previous callee c on the recursion stack and pursues the next caller of c for backward expansion. Backtracking may occur on obtaining infeasibility after either (a) forward expansion (on conjoining with SC) or (b) backward expansion (on conjoining with $CC(h_k)$). As we will see in Sec. 5, ALTER *learns* facts responsible for the current failure and uses them to avoid similar failures during future exploration.

ALTER may terminate with either (a) witness (WIT) or (b) no witness (NOWIT) or (c) an inconclusive (UNKNOWN) result. During backward exploration, if ALTER encounters an entrypoint procedure ($ENTRYPOINT(f)$) and the current ϕ is feasible, then a potential witness exists. If ϕ is skolem-free, ALTER concludes that a witness exists and returns the corresponding call context. Otherwise, ϕ may still contain skolems which cannot be expanded further, e.g., due to recursion bounds. Consequently, there may exist skipped callees which affect the feasibility of ϕ , thus making the witness spurious. In this case, ALTER returns an UNKNOWN value. Finally, if ALTER finishes exploring all callers without finding an actual witness or an UNKNOWN result, then ALTER concludes that no witness to the goal location exists. Note that obtaining an UNKNOWN value for some call context does not imply that the search is inconclusive; ALTER may go on to find an actual witness along a different call context. However, ALTER cannot infer no-witness if an UNKNOWN value is obtained for some context during exploration.

Example. Let us see how ALTER analyzes the program App1 in Fig. 11. The goal function is `A.init`, where a potential null dereference may occur at line 11 because the class `A`'s local field `this.srcs` (non-null) is shadowed by the local parameter variable `srcs`. `A.init` has two callers: `T.T` and `A.A` where `A.A` is, in turn, called by `M.M`.

<pre> ALTER(f, ϕ) if ϕ is UNSAT then \sqsubseteq return NOWIT /* Forward Expansion */ $SC := EXPANDFWD(f, \phi)$ if $(\phi \wedge SC)$ is UNSAT then \sqsubseteq return NOWIT if ENTRYPPOINT(f) then if $(\phi \wedge SC)$ has no skolems then \sqsubseteq return (WIT, nil) else \sqsubseteq return UNKNOWN </pre>	<pre> $inconcl := \text{false}$ foreach $h_k \in \text{CALLERS}(f)$ do /* Backward Expansion */ $CC(h_k) := EXPANDBWD(h_k \rightarrow f, \phi \wedge SC)$ $ans := ALTER(h, \phi \wedge SC \wedge CC(h_k))$ if $ans = (\text{WIT}, l)$ then \sqsubseteq return (WIT, $[h_k, l]$) if $ans = \text{UNKNOWN}$ then \sqsubseteq $inconcl := \text{true}$ if $inconcl$ then \sqsubseteq return UNKNOWN return NOWIT </pre>
--	---

Algorithm 1. Alternating Expansion Algorithm for Bug Detection

1. First ALTER computes a local EC ϕ for $A.\text{init}$. This $\phi := \phi_1 \wedge \phi_2$ where $\phi_1 := ((srcs_{A.\text{init}} \neq \text{null}) \wedge (this.srcs \neq \text{null})) \vee (srcs_{A.\text{init}} = \text{null})$ and $\phi_2 := (srcs_{A.\text{init}} = \text{null})$ and $srcs_{A.\text{init}}$ refers to the value of parameter $srcs$ of $A.\text{init}$. On simplifying ϕ_1 with ϕ_2 , we get $\phi := (srcs_{A.\text{init}} = \text{null})$. Because ϕ does not contain any skolems, ALTER proceeds with backward expansion along some caller, say $T.T$.
2. ALTER computes the local summary for $T.T$ and employs the call site component, (ψ, σ) for backward expansion, where the reachability condition $\psi := (srcs_{T.T} \neq \text{null} \wedge \neg sk^{ie})$ and value map $\sigma = (srcs \rightarrow srcs_{T.T})$, where $srcs_{T.T}$ refers to the value of parameter $srcs$ in $T.T$ and sk^{ie} corresponds to return value of isEmpty function. In ψ , $(srcs_{T.T} \neq \text{null})$ appears because otherwise the previous call to isEmpty will throw an exception. After expansion, we obtain $\phi := (\psi \wedge (srcs_{T.T} = \text{null}))$, which simplifies to $false$, implying search failure along $T.T$. ALTER now backtracks to try the next caller $A.A$ for $A.\text{init}$.
3. For $A.A$, the call site summary is $(true, \sigma')$ where $\sigma' := (srcs \rightarrow srcs_{A.A}, b \rightarrow b_{A.A})$. On propagation, $\phi := (srcs_{A.A} = \text{null})$, which remains satisfiable. So, ALTER expands further backwards along caller $M.M$.
4. The call site summary for $M.M$ is $(true, \sigma'')$ where $\sigma'' := (srcs \rightarrow sk^{ml}, b \rightarrow sk^{mb})$ where sk^{ml} and sk^{mb} denote the skolems corresponding to the return values of calls to makeList and makeBounds . Now, $\phi := (sk^{ml} = \text{null})$, which leads ALTER to perform forward expansion to compute the return value of makeList .
5. The side-effect summary for makeList is computed next: the summary value for the returned variable (SC_1) is $ret^{ml} := \text{alloc}(ArrayList, 2)$. Because $sk^{ml} = ret^{ml}$, we get $\phi := (\text{alloc}(ArrayList, 2) = \text{null})$ which again simplifies to $false$.

Thus, ALTER is able to show the absence of null dereference in $A.\text{init}$ by a combination of backward and forward expansion starting from the goal location in $A.\text{init}$. Note how it avoids exploration of irrelevant program regions which are in the nearby scope, i.e., functions makeBounds in $M.M$, addAll in $A.\text{init}$, isEmpty in $T.T$, add in $M.\text{makeList}$ and other callers of $M.M$ and $T.T$. Also, note how forward expansion of $M.\text{makeList}$ ensures early backtrack and avoids further backward expansion from $M.M$. The following theorem proves the correctness of ALTER.

Theorem 1. Given a goal location l , (a) if ALTER returns a witness (WIT) result then there must exist a global witness for l , and (b) if ALTER returns no-witness (NOWIT) then no global witness exists. \square

5 Learning for Efficient Expansion

Naïve alternating expansion (Sec. 4.1) may perform redundant analysis by revisiting the same callers and callees and fail repeatedly. We now present an improved ALTER algorithm for efficient exploration based on learning *caller* and *callee* invariants and employing them to prune future search. The *caller invariant* $\Omega(f)$ for a procedure f over-approximates the incoming data values from the callers of f , while the *callee invariant* $\Theta(f)$ over-approximates the return values (side-effects, in general) of the callees in f . Both these invariants are learned from expansion failures, i.e., when the constraints added due to an expansion lead to infeasibility of the error condition. Alg. 2 shows the ALTER algorithm combined with failure-driven learning of *caller* and *callee* invariants.

ALTER initializes $\Omega(f)$ and $\Theta(f)$ for all procedures f to *true* and strengthens them during exploration iteratively. The caller invariant $\Omega(f)$ is computed as disjunction of *call edge invariants* (ω) which label each incoming call edge to f . When backward expansion from f to a caller h at a call site h_k fails, i.e., $ans = (\text{NOWIT}, Inv_h)$ at location **F2** in Alg. 2 then ALTER learns a call edge invariant ω (**L2**) along the edge $h_k \rightarrow f$ using the procedure $\text{LEARN}\omega$. To this end, it splits the EC into caller- and callee-specific parts, A and B respectively, where $A \wedge B$ is infeasible. The caller-specific part, A consists of call context constraints $CC(h_k)$ and invariants Inv_h of h (usually, $\Omega(h) \wedge \Theta(h)$) which cause infeasibility. The callee-specific part, B consists of the original ϕ in f together with forward constraints SC . Note that A and B only share the input variables of f . $\text{LEARN}\omega$ now computes an *interpolant* I of A and B over the common variables of A and B such that $A \Rightarrow I$ and $I \wedge B$ is infeasible. I.e., I is an expression over input variables of f such that it over-approximates the caller constraints and is still infeasible with the error condition in f . $\text{LEARN}\omega$ now strengthens $\omega(h_k \rightarrow f)$ with I by conjoining I with the previous value of $\omega(h_k \rightarrow f)$. Then, ALTER backtracks and explores a different caller of f . Note that $\Omega(f)$ is updated when any of the call edge invariants change.

Similarly, ALTER computes (and updates) the callee invariant for f using $\text{LEARN}\Theta$ when forward expansion of ϕ from f fails (**F1**). In this case, the constraints are partitioned (**L1**) again into callee-specific (SC) and caller-specific ($\phi \wedge \Omega(f)$) parts, and an interpolant I of the two formulae is computed which over-approximates SC . The callee invariant $\Theta(f)$ is then strengthened by conjoining it with the new invariant I .

Note how both Ω and Θ are employed during exploration. Before forward expansion at location **C1**, ALTER first checks the current ϕ against the conjunction of both the invariants of f . Note that the invariants over-approximate the values from callers and callees of f . Hence, if the check with invariants is infeasible, no witness is possible on further expansion, and ALTER backtracks with NOWIT. Similarly, before backward expansion along $h_k \rightarrow f$ at location **C2**, ALTER checks ϕ against call edge invariants $\omega(h_k \rightarrow f)$, and backtracks if the check is infeasible. Lemma 1 and Theorem 2 prove the correctness of caller/callee invariants computed by Alg. 2.

² All proofs are omitted to the full version of this paper [23] due to space constraints.

INITIALLY, $\forall(h_k \rightarrow f), \omega(h_k \rightarrow f) := \text{true}$ $\forall f, \Omega(f) := \text{true}, \Theta(f) := \text{true}$ ALWAYS, $\Omega(f) := \bigvee(\omega(h_k \rightarrow f) \mid h_k \in \text{CALLERS}(f))$	[F1] if $\phi \wedge SC \wedge \Omega(f)$ is UNSAT goto [L1] if ENTRYPPOINT(f) then if $(\phi \wedge SC)$ has no skolems then return (WIT, nil) else return UNKNOWN inconcl := false foreach $h_k \in \text{CALLERS}(f)$ do [C2] if $\phi \wedge SC \wedge \omega(h_k \rightarrow f) = \text{UNSAT}$ then continue /* Backward Expansion */ $CC(h_k) := \text{EXPANDBWD}(h_k \rightarrow f, \phi \wedge SC)$ $ans := \text{ALTER}(h_k, \phi \wedge SC \wedge CC(h_k))$ if $ans = (\text{WIT}, l)$ then return (WIT, $[h_k, l]$) if $ans = \text{UNKNOWN}$ then inconcl := true [F2] if $ans = (\text{NOWIT}, Inv_h)$ then [L2] $\text{LEARN}\omega(h_k \rightarrow f, CC(h_k) \wedge Inv_h, \phi \wedge SC)$ if inconcl then return UNKNOWN [L1] $\text{LEARN}\Theta(f, SC, \phi \wedge \Omega(f))$ [E] return (NOWIT, $\Theta(f) \wedge \Omega(f)$)
begin $I := \text{INTERPOLANT}(a, b)$ $\omega(h_k \rightarrow f) := \omega(h_k \rightarrow f) \wedge I$ begin $I := \text{INTERPOLANT}(a, b)$ $\Theta(f) := \Theta(f) \wedge I$ ALTER (f, ϕ) [S] if ϕ is UNSAT then return (NOWIT, true) [C1] if $\phi \wedge \Theta(f) \wedge \Omega(f)$ is UNSAT then return (NOWIT, $\Theta(f) \wedge \Omega(f)$) /* Forward Expansion */ $SC := \text{EXPANDFWD}(f, \phi)$	if $\phi \wedge \Theta(f) \wedge \Omega(f)$ is UNSAT goto [L1] if ENTRYPPOINT(f) then if $(\phi \wedge SC)$ has no skolems then return (WIT, nil) else return UNKNOWN inconcl := false foreach $h_k \in \text{CALLERS}(f)$ do [C2] if $\phi \wedge SC \wedge \omega(h_k \rightarrow f) = \text{UNSAT}$ then continue /* Backward Expansion */ $CC(h_k) := \text{EXPANDBWD}(h_k \rightarrow f, \phi \wedge SC)$ $ans := \text{ALTER}(h_k, \phi \wedge SC \wedge CC(h_k))$ if $ans = (\text{WIT}, l)$ then return (WIT, $[h_k, l]$) if $ans = \text{UNKNOWN}$ then inconcl := true [F2] if $ans = (\text{NOWIT}, Inv_h)$ then [L2] $\text{LEARN}\omega(h_k \rightarrow f, CC(h_k) \wedge Inv_h, \phi \wedge SC)$ if inconcl then return UNKNOWN [L1] $\text{LEARN}\Theta(f, SC, \phi \wedge \Omega(f))$ [E] return (NOWIT, $\Theta(f) \wedge \Omega(f)$)

Algorithm 2. ALTER with learning caller Ω and callee Θ invariants

Lemma 1. The following invariants hold in Alg. 2 (a) $(\Omega(h) \wedge \Theta(h)) \Rightarrow Inv_h$ (b) $(CC(h_k) \wedge Inv_h \wedge \phi \wedge SC)$ is unsatisfiable at **L2**, $SC \wedge \phi \wedge \Omega(f)$ is unsatisfiable at **L1**. (c) $(\Omega(h) \wedge \Theta(h) \wedge CC(h_k)) \Rightarrow \omega(h_k \rightarrow f)$

Theorem 2. Given a procedure f , (a) the caller invariant $\Omega(f)$ over-approximates the incoming data values from all the callers of f and (b) the callee invariant $\Theta(f)$ over-approximates the side-effects of the callees of f .

Proofs of Non-violation. If the analysis returns NOWIT, then the set of caller and callee invariants constitute a proof for absence of violation in the goal function g . In other words, we can conclude that null dereference is not possible at the goal location by using the caller $\Omega(g)$ and callee $\Theta(g)$ invariants for g . These invariants are obtained, in turn, from the invariants of other functions in the scope of the analysis. The undecidability of program analysis implies we cannot always obtain such a proof; however, in practice, we obtain proofs for absence of null dereference in several of our benchmarks. Note that the learned facts can be reused to improve search when checking multiple goals in the same application (cf. Sec. 6). Further, they are useful for re-validation across upgrades of an application; we leave investigating the usefulness of learned facts during incremental verification to a future work.

5.1 Examples Illustrating the Learning Algorithm

Example 1. Consider the program and its call graph in Fig. 4. Suppose the functions q and s are the entry points and the call to $error()$ in p is a null dereference. Fig. 6 shows the ECs and invariants computed by ALTER on this program, starting with $true$ for all caller and callee invariants. The initial EC is $\phi_0 := (x < 10)$ in p .

1. ALTER first propagates ϕ_0 to caller q at site q_2 to get ϕ_1 (cf. Fig. 6(a)). Then, it expands forward sk^t in ϕ_1 to obtain ϕ_2 , which is infeasible. ALTER learns the callee invariant $\Theta(q)$ from this failure (location [L1] in Algo. 2): it splits ϕ_2 into $A := SC_q \equiv (sk^t = y * 2)$ and $B := \phi_1 \wedge \Omega(q) \equiv (y > 6 \wedge sk^t < 10) \wedge (true)$ and computes interpolant $\Theta_0 = (sk^t \geq y * 2)$ (cf. Fig. 6(b)). Then, it updates $\Theta(q) := \Theta_0$ and backtracks to p .
2. In p , ALTER now continues to learn a call edge invariant $\omega(q_2 \rightarrow p)$ ([L2] in Alg. 2) based on the previous failure. It partitions ϕ_2 into $A := \Omega(q) \wedge \Theta(q) \wedge CC(q_2) \equiv (true) \wedge (sk^t \geq y * 2) \wedge (y > 6 \wedge x = sk^t)$ and $B := \phi_0$, computes interpolant $\omega_1 := (x \geq 14)$ and updates $\omega(q_2 \rightarrow p) := \omega_1$ (Fig. 6(b)). Now, ALTER propagates ϕ_0 back to next caller r of p at call site r_1 as ϕ_3 and then to s at s_1 as ϕ_4 . Here, ϕ_4 is infeasible. Thus, ALTER backtracks to r and learns $\omega(s_1 \rightarrow r) = (v \geq 10)$.
3. Now, it propagates ϕ_3 to q from r and obtains ϕ_5 which is satisfiable. However, when ϕ_5 is conjoined with $\Theta(q)$, it becomes infeasible [C1]. Therefore, ALTER uses $\Theta(q)$ learned from previous failure in q to backtrack to r and avoid multiple forward expansions of t in q . On backtracking, it learns $\omega(q_3 \rightarrow r) := (u \leq v - 7)$ and updates $\Omega(r) := \omega_2 \vee \omega_3 \equiv (u \leq v - 7) \vee (v \geq 10)$.
4. As ALTER failed on all callers of r , it backtracks to p and learns $\omega(r_1 \rightarrow p) := \omega_4 \equiv (x \geq 11)$. ALTER now tries the next caller r_2 of p to obtain ϕ_7 , which is feasible. Next, all callers of r are tried: ALTER first checks ϕ_7 against current call edge invariant

	INV	A	B	INTERPOLANT
ϕ_0	INITIAL EC	$(x < 10)$		
ϕ_1	EXPANDBWD(ϕ_0, q_2)	$(y > 6 \wedge sk^t < 10)$	SAT	
ϕ_2	EXPANDFWD(ϕ_1, q)	$(y > 6 \wedge y * 2 < 10)$	SAT	
ϕ_3	EXPANDBWD(ϕ_0, r_1)	$(u < 10 \wedge u > v)$	UNSAT	Θ_0, ω_1
ϕ_4	EXPANDBWD(ϕ_3, s_1)	$(c < 10 \wedge c > 10)$	UNSAT	ω_2
ϕ_5	EXPANDBWD(ϕ_3, q_3)	$(y > 6 \wedge y < 10 \wedge y > sk^t)$	SAT	
ϕ_6	CHK($\phi_5, \Theta(q)$)	$(y > 6 \wedge y < 10 \wedge y > sk^t) \wedge (sk^t \geq y * 2)$	UNSAT	ω_3, ω_4
ϕ_7	EXPANDBWD(ϕ_0, r_2)	$(v < 10 \wedge u \leq v)$	SAT	
ϕ_8	CHK($\phi_7, \Omega(r)$)	$(v < 10 \wedge u \leq v) \wedge (u \leq v - 7 \vee v \geq 10)$	SAT	
ϕ_9	CHK($\phi_7, \omega(s_1 \rightarrow r)$)	$(v < 10 \wedge u \leq v) \wedge (v \geq 10)$	UNSAT	-
ϕ_{10}	CHK($\phi_7, \omega(q_3 \rightarrow r)$)	$(v < 10 \wedge u \leq v) \wedge (u \leq v - 7)$	SAT	
ϕ_{11}	EXPANDBWD(ϕ_7, q_3)	$(y > 6 \wedge sk^t < 10 \wedge y \leq sk^t)$	SAT	
ϕ_{12}	CHK($\phi_{11}, \Theta(q)$)	$(y > 6 \wedge sk^t < 10 \wedge y \leq sk^t) \wedge (sk^t \geq y * 2)$	UNSAT	ω_5, ω_6

	INV	A	B	INTERPOLANT
Θ_0	$\Theta(q)$	$(sk^t = y * 2)$	$(y > 6 \wedge sk^t < 10)$	$sk^t \geq y * 2$
ω_1	$\omega(q_2 \rightarrow p)$	$(sk^t \geq y * 2) \wedge (y > 6 \wedge x = sk^t)$	$(x < 10)$	$x \geq 14$
ω_2	$\omega(s_1 \rightarrow r)$	$(u = c \wedge v = 10)$	$(u < 10 \wedge u > v)$	$v \geq 10$
ω_3	$\omega(q_3 \rightarrow r)$	$(sk^t \geq y * 2) \wedge (y > 6 \wedge u = y \wedge v = sk^t)$	$(u < 10 \wedge u > v)$	$u \leq v - 7$
ω_4	$\omega(r_1 \rightarrow p)$	$(u \leq v - 7 \vee v \geq 10) \wedge (u > v \wedge x = u)$	$(x < 10)$	$x \geq 11$
ω_5	$\omega(q_3 \rightarrow r)$	$(sk^t \geq y * 2) \wedge (y > 6 \wedge u = y \wedge v = sk^t)$	$(v < 10 \wedge u \leq v)$	$v \geq 14$
ω_6	$\omega(r_2 \rightarrow p)$	$((u \leq v - 7 \wedge v \geq 14) \vee (v \geq 10)) \wedge (u \leq v \wedge x = v)$	$(x < 10)$	$x \geq 10$

(b)

Fig. 6. Illustration of the Learning Algorithm for Program P in Fig. 4

value $\omega(s_1 \rightarrow r)$, which is infeasible; it next tries $\omega(q_3 \rightarrow r)$, which is feasible. So, ϕ_7 propagates back to q as ϕ_{11} . In q , however, ϕ_{11} becomes unsatisfiable with $\Theta(q)$, forcing backtrack to r while updating $\omega(q_3 \rightarrow r) := \omega_3 \wedge \omega_5$ and $\Omega(r) := \omega_2 \vee (\omega_3 \wedge \omega_5)$. Because all callers of r are explored, ALTER further backtracks to p while updating $\omega(r_2 \rightarrow p) := \omega_6$. Finally, no feasible paths to error in p exist; ALTER returns NOWIT.

Example 2. Recall the example in Fig. 3 where ALTER redundantly explores callers runA and runB multiple times. Learning solves this problem: after failing with context $\text{runA} \rightarrow \text{foo}_1 \rightarrow \text{bar}$, ALTER labels edge $\text{runA} \rightarrow \text{foo}$ with predicate $\omega_1 = (c_{\text{foo}} \neq \text{null})$. Similarly, edge $\text{runB} \rightarrow \text{foo}$ is labeled with ω_1 . Because both callers of foo have been explored, ALTER now computes a call invariant $\Omega_1 = (c_{\text{foo}} \neq \text{null})$ for foo by disjoining the incoming edge invariants. This invariant helps to prune backward search in the second iteration: the EC ($c_{\text{foo}} = \text{null}$) for context $\text{foo}_2 \rightarrow \text{bar}$ is unsatisfiable immediately on conjoining with Ω_1 . Hence, ALTER avoids the redundant exploration of runA and runB for the second call to bar .

6 Evaluation

We implemented the ALTER algorithm using the WALA framework for analyzing Java programs and applied it to validate the null dereference warnings produced by FindBugs [10], in a manner similar to the earlier Snufflebug work [4], where these benchmarks were validated using weakest precondition computation. We considered three open-source Java benchmarks, *apache-ant*(v1.7), *batik*(v1.6) and *tomcat*(v6.0.16), having LoC 88k, 157k and 163k, respectively.

Our analysis finds global witnesses with respect to a set of given entrypoints; we initialized the set of entrypoints to all public methods without any callers. Procedure summarization is done on-demand during forward/backward expansion. We used the CVC3 solver [2] to check the satisfiability of ECs and the MathSAT5 solver [7] to compute interpolants. A coarse mod-ref analysis is performed on the call graph in the beginning to compute side-effects. Extensive formula simplification is performed in ALTER using a pre-defined set of rewrite rules [4]. Forward expansion involves recursive expansion of skolems as the predominant strategy, with feedback driven expansion for virtual call skolems [4] (cf. [23]). We also tried lazy expansion strategies [1]; however, recursive forward expansion outperforms lazy expansion in most cases.

We designed a set of experiments: First, we compare ALTER with a non-alternating version NOALT which performs forward expansion only after backward expansion terminates at an entrypoint. Next, we evaluate the impact of learning. Finally, since we consider Snufflebug (SB) to be an ancestor of ALTER (they do share significant amount of code), we also compare the end-to-end performance of ALTER with SB.

Fig. 7 shows the ALTER results on a set of dereference checks for above benchmarks (each check corresponds to a single warning reported by FindBugs). All the benchmarks contain a combination of witness and no-witness instances.³ We show only the actual analysis run times; the initial call graph and mod-ref computation times are excluded. The results also show the number of functions summarized by ALTER and the maximum error depth for the checks: the alternating expansion by ALTER succeeds in finding a witness or showing its absence by analyzing a small set of functions around the goal.

³ The table excludes Snufflebug benchmarks on which either ALTER reported inconclusive (due to recursion), did not finish or the run times of both the tools were very small.

ALTER outperforms NOALT on most benchmarks: although NOALT performs similar to ALTER for bugs where entrypoints are closer to the goal function, it times-out on deeper goal functions. For example, NOALT performs poorly on `tomcat14` because it redundantly explores a much longer call context that does not lead to an error, and wastes resources performing many redundant forward expansions. In contrast, ALTER finds a call context of depth 6 that leads to a witness. This shows the advantage of alternating expansion clearly: expanding forward before backward avoids exploring long redundant contexts and helps obtain smaller scopes on our benchmarks.

Benchmark	WIT?	T(SB)	#FS(SB)	T(NOALT)	MaxD(NOALT)	#FS(NOALT)	T(ALTER)	MaxD(ALTER)	#FS(ALTER)
ant3	Y	>300	>154	0.6	0	1	0.6	0	1
ant4	N	4.17	102	1.9	1	2	1.21	1	2
ant5	N	2.7	66	0.8	0	1	0.87	0	1
batik2	Y	7.6	33	1.0	2	4	0.9	2	4
batik5	Y	11.5	25	18.3	23	91	5.1	9	26
batik7	N	3.5	37	> 300	> 38	> 100	1.3	3	6
batik8	N	4.5	30	2.4	1	3	2.5	1	3
batik9	Y	48.7	89	6.79	3	21	5.6	2	14
batik10	Y	3.8	88	1.7	2	4	1.8	2	4
tomcat9	N	114	26	> 300	> 16	> 74	2.8	0	7
tomcat10	Y	4.9	26	4.1	4	17	3.7	4	17
tomcat11	Y	19.64	7	0.8	0	3	0.86	0	3
tomcat12	N	> 300	>50	0.94	1	2	0.9	0	2
tomcat14	Y	6.1	26	> 300	> 17	> 55	1.778	6	7

Fig. 7. Comparison of Snugglebug (SB), NOALT and ALTER on Java benchmarks. WIT? = witness or not. All times in seconds. MaxD denotes the length of longest call context to the goal function during exploration, #FS denotes the total number of functions summarized during each analysis.

Benchmark	#Goals	Time(NL)	Time(L)	Time(Itp)	Edge(NL)	Edge(L)	LrnReUse	LrnEdge	LrnUpdts
ant3	9	4.013	3.996	0	8	8	0	3	3
ant4	6	1.377	1.527	0.214	3	3	0	2	3
ant5	7	1.302	1.36	0	0	0	0	0	0
batik2	20	1.349	1.589	0.183	4	4	0	1	2
batik7	23	9.319	9.529	0.546	50	41	9	6	7
batik8	24	9.113	9.179	0.461	9	9	0	2	3
batik9	32	8.508	9.879	0.931	31	31	0	8	8
batik10	20	2.558	2.45	0.306	13	9	2	2	3
tomcat9	54	24.511	26.736	0.209	68	68	0	2	2
tomcat10	33	9.193	10.542	3.203	105	39	3	23	23
tomcat11	16	2.519	2.573	0	0	0	0	0	0
tomcat12	18	4.771	4.949	0	16	16	0	0	0
tomcat14	4	2.24	1.934	0.23	17	9	4	4	4

Fig. 8. Evaluation of learning in ALTER on Java benchmarks. Time : Time for analysis. L-learning, NL-No learning, Itp : Interpolant generation during learning. Edge : Number of edges explored in callgraph. LrnReUse : Number of times previous learning helped in backtracking. LrnEdges : Number of edges with learning. LrnUpdt : Total number of learning updates. *batik5* (multiple goals) does not finish because of bugs in our tool.

Fig. 8 shows the impact of learning on alternating expansion, both in terms of the run-time and the edges explored during backward expansion: our experiments primarily focused on learning and reusing caller invariants. Instead of analyzing a single goal, we collect multiple null dereferences from the goal function and analyze them in sequence. This allows the successive runs to take advantage of previously learned invariants. The results show that learning invariants indeed reduces the number of call graph edges re-explored (Edge(L) vs Edge(NL)) by reusing invariants learned earlier. In some cases, e.g., `tomcat10`, the number of edges explored reduces by almost two-thirds. In contrast, the run-time benefits depend on how effectively the invariants are reused: if there is plenty of reuse, the ALTER run-time is lower. However, if the overhead of computing invariants is much larger than the reduction due to reuse, ALTER is slower with learning. For example, although ALTER explores much fewer edges in `tomcat10`, the time taken for interpolant generation is also large (3.203 seconds), which annuls the benefits of learning. However, in such cases, learning provides proofs (at a small cost) which we believe amounts to long term benefits, e.g. during regression testing across upgrades. We believe the results will improve further by employing a single solver for both checking infeasibility and interpolant generation (we used two solvers because we wanted to reuse our existing stable interface to CVC3) and compute interpolants in-memory.

Finally, Fig 7 also shows that ALTER consistently finishes faster than SB. In particular, on `ant3` and `tomcat12`, ALTER finishes quickly while Snufflebug times out (5 minutes). ALTER and SB are architecturally very different and it is difficult to narrow down the cause for the large performance difference to a single factor. One factor is that ALTER computes and reuses local summaries as opposed to SB which may re-analyze procedures for different call contexts. Another factor is that intraprocedurally, ALTER merges symbolic states at join points, whereas SB does not, due to which it needs to propagate a large number of different formulae through a control-flow graph. Finally, SB does not implement alternating scope expansion or learning.

7 Related Work

Loginov et. al. [16] present a closely-related analysis that expands the scope around the goal function in a breadth-first fashion, iteratively analyzing larger scopes until it finds a witness. Breadth-first expansion was also used in the work of Ma et al. [17], which combines forward and backward exploration for testing. In some cases, a strict breadth-first strategy may lead to excessive analysis of irrelevant code, e.g., when the goal function has many callees irrelevant to the property. ALTER uses a more sophisticated alternating search strategy to avoid analyzing such irrelevant code. The probabilistic analysis of Gulwani and Jovic also combines forward and backward exploration [8], but their work does not focus on handling of procedure calls in large programs.

Scope-bounded analysis in DC2 [11] bounds the program scope and computes environment (caller) constraints and (callee) function stubs for the procedures outside the scope using a light-weight whole program analysis. However, scope bounding is performed manually, without automatic scope expansion. ALTER could also be extended to exploit separately-computed caller and callee invariants. Snufflebug (SB) [4] tries to detect bugs by performing backward weakest precondition computation on the inter-procedural control flow graph. Unlike ALTER, SB may re-analyze functions for different postconditions, and it does not learn facts from failed backward propagation.

Structural abstraction techniques [1][24][22] focus on heuristics for lazy forward expansion. CORRAL [15] performs efficient forward expansion in a stratified manner (a variant of structural abstraction/refinement) together with selective variable abstraction. CORRAL also uses separately-computed invariants to improve search. Unlike ALTER, these techniques have no backward expansion, helpful for deep goal functions, and no automated invariant learning to avoid redundant re-analysis.

Our learning technique is influenced by the DPLL paradigm, in general, and by *lazy annotation* [18], in particular. The latter learns program annotations from failed explorations during path-enumeration-based analysis but starts from the *main* routine, which may make it hard to locate bugs in deep callees. Also, it performs basic block-level expansion and fine-grained learning at the intra-procedural level, which may aggravate path explosion when finding long inter-procedural witnesses. In contrast, ALTER employs local procedure summaries, which avoid re-analysis of procedures as well as both intra- and inter-procedural path explosion. By expanding a whole procedure in one step and learning constraints at procedure interfaces, ALTER is able to focus on inter-procedural exploration without being distracted by repeated intra-procedural analysis.

The SMASH tool [5] employs a combination of *may* and *must* summaries obtained from predicate abstraction and directed symbolic execution, respectively, to avoid redundant re-analysis. Both these summaries are approximations (over- and under-, respectively) of callee side-effects and are useful for forward expansion. Here, we propose to compute caller invariants to improve backward expansion besides employing callee invariants for forward search. Call invariants proposed by Lahiri and Qadeer [14] may be seen as a restricted form of callee invariants which capture the memory footprint unchanged by a procedure.

More broadly, many recent systems for verification and bug detection have been based on predicate abstraction (e.g., BLAST [9] and CPACHECKER [3]). Predicate-abstraction approaches suffer from expensive predicate image computation and, typically, cannot recover from irrelevant refinements. In contrast, ALTER performs a sort of lazy annotation [18] at procedure boundaries, which is able to generalize from invariants specific to a particular call context. Also, while predicate abstraction has worked well on certain kinds of programs (e.g. programs arising from the device-driver domain), it has not been shown to work well on general object-oriented programs. A key challenge with OO programs is heavy use of heap structures, which makes the predicate space that can adequately abstract a program difficult to identify.

8 Conclusions

We proposed a new scalable method to detect inter-procedural bugs using a focused, alternating backward and forward expansion strategy, starting from the goal function. The method iteratively explores the call contexts of the goal function and the callees thereof in an alternating manner, backtracks from infeasible contexts, and learns caller-/callee invariants from failed explorations to prune future search. We demonstrated the effectiveness of our method on large open-source Java programs in terms of faster run times and lesser analysis scopes. In future, we will investigate better forward expansion strategies and improve reuse and management of learned facts.

References

1. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007)
2. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
3. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
4. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: PLDI, pp. 363–374 (2009)
5. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL, pp. 43–56 (2010)
6. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI, pp. 213–223 (2005)
7. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8, 1–27 (2012)
8. Gulwani, S., Jovic, N.: Program verification as probabilistic inference. In: POPL (2007)
9. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002 (2002)
10. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: OOPSLA Companion (2004)
11. Ivancic, F., Balakrishnan, G., Gupta, A., Sankaranarayanan, S., Maeda, N., Tokuoka, H., Imoto, T., Miyazaki, Y.: DC2: A framework for scalable, scope-bounded software verification. In: ASE, pp. 133–142 (2011)
12. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
13. Kölbl, A., Pixley, C.: Constructing efficient formal models from high-level descriptions using symbolic simulation. IJPP 33(6), 645–666 (2005)
14. Lahiri, S.K., Qadeer, S.: Call Invariants. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 237–251. Springer, Heidelberg (2011)
15. Lal, A., Qadeer, S., Lahiri, S.: Corral: A Solver for Reachability Modulo Theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
16. Loginov, A., Yahav, E., Chandra, S., Fink, S., Rinetzky, N., Nanda, M.G.: Verifying dereference safety via expanding-scope analysis. In: ISSTA, pp. 213–224 (2008)
17. Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed Symbolic Execution. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 95–111. Springer, Heidelberg (2011)
18. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
19. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL, pp. 49–61. ACM, NY (1995)
20. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, vol. 5, pp. 189–234. Prentice Hall (1981)
21. Sinha, N.: Symbolic program analysis using term rewriting, generalization. In: FMCAD (2008)
22. Sinha, N.: Modular bug detection with inertial refinement. In: FMCAD (2010)
23. Sinha, N., Singhania, N., Chandra, S., Sridharan, M.: Scalable bug detection via alternating scope expansion and pertinent scope learning. IBM Technical Report RI12003 (2012)
24. Taghdiri, M., Jackson, D.: Inferring specifications to detect errors in code. Autom. Softw. Eng. 14(1), 87–121 (2007)

A Complete Method for Symmetry Reduction in Safety Verification

Duc-Hiep Chu^{*} and Joxan Jaffar

National University of Singapore
`{hiepcd,joxan}@comp.nus.edu.sg`

Abstract. Symmetry reduction is a well-investigated technique to counter the state space explosion problem for reasoning about a concurrent system of similar processes. Here we present a general method for its application, restricted to verification of safety properties, but *without* any prior knowledge about global symmetry. We start by using a notion of *weak symmetry* which allows for more reduction than in previous notions of symmetry. This notion is relative to the target safety property. The key idea is to perform symmetric transformations on *state interpolation*, a concept which has been used widely for pruning in SMT and CEGAR. Our method naturally favors “quite symmetric” systems: more similarity among the processes leads to greater pruning of the tree. The main result is that the method is *complete* wrt. weak symmetry: it only considers states which are not weakly symmetric to an already encountered state.

1 Introduction

Symmetry reduction is a well-investigated technique to counter the state space explosion problem when dealing with concurrent systems whose processes are similar. In fact, traditional symmetry reduction techniques rely on an idealistic assumption that processes are *indistinguishable*. Because this assumption excludes many realistic systems, there is a recent trend [7, 12, 14, 15] to consider systems of non-identical processes, where the processes are *sufficiently similar* that the original gains of symmetry reduction can still be accomplished. However, this necessitates an intricate step of detecting symmetry in the state exploration.

We start by considering an intuitive notion of symmetry, which is based on a standard adaptation of the notion of bisimilarity. We say two states s_1 and s_2 are symmetric if there is a “permutation” π such that $s_2 = \pi(s_1)$, and if each successor state of s_1 can be matched (via π) with a unique successor state of s_2 while at the same time each successor state of s_2 can be matched (via π^{-1}) with a unique successor state of s_1 . In safety verification, we further require that s_1 is safe iff s_2 is safe.

We refer to this notion as *strong symmetry*. We mention that all recent works which deal with heterogeneous systems (processes are not necessarily identical)

^{*} This author is supported by NUS Graduate School for Integrative Sciences and Engineering.

have the desire to capture this type of symmetry in the sense that they attempt, though not quite successfully, to consider only states which are *not* strongly symmetric to any already encountered state.

In this paper, we present a general approach to symmetry reduction for safety verification of a finite multi-process system, defined parametrically, without any prior knowledge about its global symmetry. In particular, we explicitly explore all possible interleavings of the concurrent transitions, while applying pruning on “symmetric” subtrees. We now introduce a new notion of symmetry: *weak symmetry*. Informally, this notion weakens the notion of permutation between states so that *the program counter* is the paramount factor in consideration of symmetry. In contrast, values of program variables are used in consideration of strong symmetry. The main result is that our approach is *complete* wrt. weak symmetry: it only considers states which are not weakly symmetric to an already encountered state.

More specifically, we address the state explosion problem by employing *symbolic learning* on the search tree of all possible interleavings. Specifically, our work is based on the concept of interpolation. Here, interpolation is essentially a form of *backward learning* where a completed search of a *safe* subtree is then formulated as a recipe for pruning (every state/node is a root associated to some subtree). There are two key ideas regarding our learning technique: First, each learned recipe for a node not only can be used to prune other nodes having the same future (same program point), but also can be *transferred* to prune nodes that having *symmetric* futures (symmetric program points). Second, each recipe discovered by a node will be conveyed back to its ancestors, which gives rise to pruning of *larger* subtree. Another important distinction is that our method learns *symbolically* with respect to the safety property and the interleavings. In Section 5, we will confirm the effectiveness of our method experimentally on some classic benchmarks.

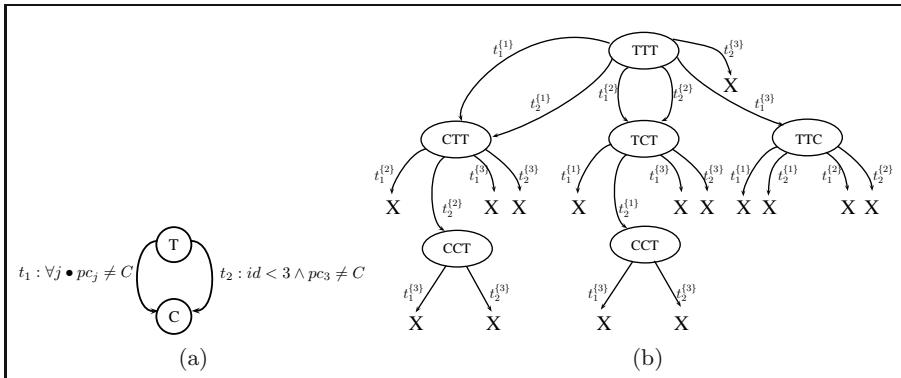


Fig. 1. (a) Modified 3-process reader-writer (b) Full interleaving tree

We conclude this subsection with two examples in order to demonstrate strong and weak symmetry. First we borrow with modification from [14][15] wherein are

two “reader” processes (indices 1, 2) and one “writer” process (index 3). We denote by C and T the local process states which indicate entering the critical section and in a “trying” state, respectively. See Figure 2(a). Note that pc_j is the local control location of process j and for each process, id is its *process identifier*. These concepts will be defined more formally in Section 2.

For each process, there are two transitions from T to C. The first, t_1 , is executable by any process provided that no process is currently in its critical section ($\forall j \bullet pc_j \neq C$). The second, t_2 , is however available to only readers ($id < 3$), and the writer must be in a non-critical local state $pc_3 \neq C$. This example shows symmetry between the reader processes, but because of their priority over the writer, we do not have “full” symmetry [14].

Figure 2(b) shows the full interleaving tree. Transitions are labelled with superscripts to indicate the process to which that transition is associated. *Infeasible* transitions are arrows ending with crosses. Note that nodes CTT and TCT are strongly symmetric, but neither is strongly symmetric with TTC.

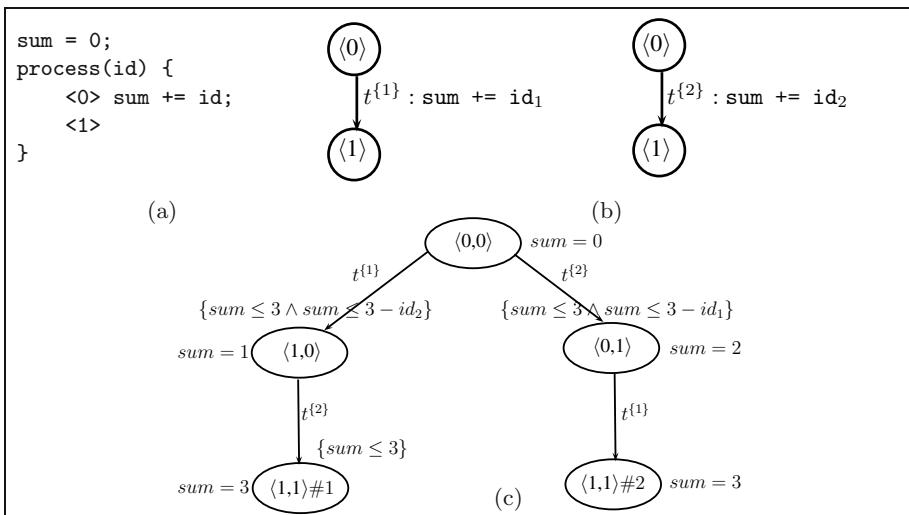


Fig. 2. (a) Sum-of-ids system (b) Its 2-process concretization (c) Full interleaving tree

Our second example is the system in Figure 2(a). Initially, the shared variable `sum` is set to 0. Each process increments `sum` by the amount of its process identifier, namely `id`. The local transition systems for process 1 and process 2 are shown in Figure 2(b). The full interleaving tree is shown in Figure 2(c).

Let π be the function swapping the indices of the two processes. We can see that the subtrees rooted at states $\langle (1, 0); sum = 1 \rangle$ and $\langle (0, 1); sum = 2 \rangle$ share the same shape. However, due to the difference in the value of shared variable `sum`, strong symmetry does not apply (in fact, any top-down technique, such as [14][15][12], cannot avoid exploring the subtree rooted at $\langle (0, 1); sum = 2 \rangle$, even

if the subtree rooted at $\langle\langle 1, 0 \rangle; sum = 1 \rangle$ has been traversed and proved to be safe).

There is however a *weaker* notion of symmetry that does apply. We explain this by outlining our own approach, whose key feature is the computation of an *interpolant* [10] for a node, by a process of backward learning. Informally, this interpolant represents a *generalization* of the values of the variables such that the traversed tree has a similar transition structure, and also remains safe. In the example, we require the safety property $\psi \equiv sum \leq 3$ at every state, and interpolants are shown as formulas inside curly brackets.

Using precondition propagation, the interpolant for state $\langle\langle 1, 1 \rangle; sum = 3 \rangle$ is computed as $sum \leq 3$, and the interpolant for state $\langle\langle 1, 0 \rangle; sum = 1 \rangle$ is computed as $\phi_{\langle 1, 0 \rangle} \equiv sum \leq 3 \wedge sum \leq 3 - id_2$. Using this, we can infer that $\phi_{\langle 0, 1 \rangle} \equiv sum \leq 3 \wedge sum \leq 3 - id_1$ (obtained by applying π on $\phi_{\langle 1, 0 \rangle}$) is a sound interpolant for program point $\langle 0, 1 \rangle$. As $\langle\langle 0, 1 \rangle; sum = 2 \rangle \models \phi_{\langle 0, 1 \rangle}$, the subtree rooted at $\langle\langle 0, 1 \rangle; sum = 2 \rangle$ can be pruned.

1.1 Related Work

Symmetry reduction has been extensively studied, e.g. [5, 2, 8, 6]. Traditionally, symmetry is defined as a transition-preserving equivalence, where an automorphism π , other than being a bijection on the reachable states, also satisfies that (s, s') is a transition iff $(\pi(s), \pi(s'))$ is. There, this type of symmetry reduction is enforced by *unrealistic* assumptions about indistinguishable processes. As a result, it does not apply to many systems in practice.

One of the first to apply symmetry reduction strategies to “approximately symmetric” systems is [7], defining notions of *near* and *rough* symmetry. Near and rough symmetry is then generalized in [4] to *virtual symmetry*, which still makes use of the concept of bisimilarity for symmetry reduction. Though bisimilarity enables full μ -calculus model checking, the main limitation of these approaches is that they exclude many systems, where bisimilarity to the quotient is simply not attainable. Also, these approaches work only for the verification of *fully symmetric properties*. No implementation is provided.

The work [12] allows arbitrary divergence from symmetry, and accounts for this divergence initially by conservative optimism, namely in the form of symmetric “super-structure”. Specifically, transitions are added to the structure to achieve symmetry. A *guarded annotated quotient* (GAQ) is then obtained from the super-structure, where added transitions are marked. This approach works well for programs with syntactically specified static transition priority. However, in general, the GAQ needs to be *unwound* frequently to compensate for the loss in precision (false positive due to added transitions). This might affect the running time significantly as this method might need to consider many combinations of transitions which do not belong to the original structure.

In comparison with our technique, this method has a clear advantage that it can handle arbitrary CTL* property. Nevertheless, our technique is more efficient both in space and time. Our technique is required to store an interpolant for each non-subsumed state, whereas in [12], a quotient edge might require multiple

annotations. Furthermore, ours does not require a costly preprocessing of the program text to come up with a symmetric super-structure. Also, extending [12] to symbolic model checking does not seem possible.

The most *recent* state-of-the-art regarding symmetry reduction, and also closest to our spirit, is the *lazy approach* proposed by [14,15]. Here only safety verification is considered. This approach does not assume any prior knowledge about (global) symmetry. Indeed, they initially and lazily ignore the potential lack of symmetry. During the exploration, each encountered state is annotated with information about how symmetry is violated along the path leading to it. The idea is that more similarity between component processes entails more compression is achieved.

In summary, the two main related works which are not restricted *a priori* on global symmetry are [12] and [14]. That is, these works allow the system to use process identifiers and therefore do not restrict the behaviors of individual processes. This is not the case with the previously mentioned works.

These works, [12] and [14], can be categorized as *top-down* techniques. Fundamentally, they look at the syntactic similarities between processes, and then come up with a reduced structure where symmetric states/nodes are merged into one abstract node. When model checking is performed, an abstract node might be concretized into a number of concrete nodes and each is checked one by one ([12] handles that by unwinding). For them, two symmetric parental nodes are not guaranteed to have correspondingly symmetric children. For us, by backward learning, we *ensure* that is the case. Consequently, and most importantly, they do not exponentially improve the runtime, only compress the state space.

Consider again the first example above (Figure 1). A top-down approach will consider TTC as a “potentially” symmetric state of CTT, and all three states CTT, TCT, and TTC are merged into one abstract state. While having compaction, it is not the case that the search space traversed is of this compact size. As a non-symmetric state (TTC) is merged with other mutually symmetric states (CTT and TCT), in generating the successor abstract state, the parent abstract state is required to be concretized and both transitions $t_2^{\{2\}}$ (emanating from CTT) and transition $t_2^{\{1\}}$ (emanating from TCT) are considered (in fact, infeasible transition $t_2^{\{3\}}$ is also considered). In summary, compaction may not lead to any reduction in the search space.

We finally mention that we consider only safety properties because we wish to employ abstraction in the search process. And it is precisely a judicious use of abstraction that enables us to obtain more pruning in comparison with prior techniques. We prove this in principle by showing that we are *complete* wrt. weak symmetry, and we demonstrate this experimentally on some classic benchmarks.

2 Preliminaries

We consider a parametrically defined n -process system, where n is fixed. In accordance with standard practice in works on symmetry, we assume that the domain of discourse of the program variables is *finite* so as to guarantee termination of

the search process of the underlying transition system. Infinite domains may be accommodated by some use of abstraction, as we show in one benchmark example below.

We employ the usual syntax of a deterministic imperative language, and communication occurs via shared variables. Each process has a unique and predetermined *process identifier*, and this is denoted parametrically in the system by the special variable `id`. For presentation purpose, the concrete value of `id` for each individual process ranges from 1 to n . We note that the variable `id` cannot be changed. Even though the processes are defined by one parameterized system, their dynamic behaviors can be arbitrarily different. This would depend on how `id` is expressed in the parameterized system. Finally, we also allow a blocking primitive `await(b) s;` where `b` is a boolean expression and `s` is an *optional* program statement.

Consider the 2-process parameterized system in Figure 3(a). Note the (local) program points in angle brackets. Figure 3(b) “concretizes” the processes explicitly. Note the use in the first process of the variable id_1 which is not writable in the process, and whose value is 1. Similarly for id_2 in the other process.

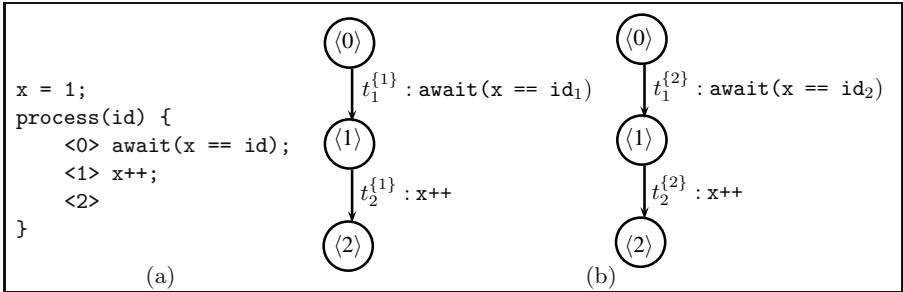


Fig. 3. (a) A parameterized system (b) Its 2-process concretization

In general, where P_i ($1 \leq i \leq n$) is a process, let V_i be its local variables and V_{shared} be the shared variables of entire system. We note here that V_i does not include the special local variables which represent the process identifiers. Let $pc_i \in V_i$ be a special variable represent the local program counter, and the tuple $\langle pc_1, pc_2 \dots, pc_n \rangle$ represent the global program point. Let $State$ be the set of all global states of the given program where $s_0 \in State$ is the initial state. A state $s \in State$ comprises of three parts: its *program point* $pc(s)$, which is a tuple of local program counters, its *valuation* over the program variables $\text{val}(s)$, and its valuation over the process identifiers pids . In other words, we denote a state s by $\langle pc(s); \text{val}(s); \text{pids} \rangle$. Note that all states from the same parameterized system share the same valuation of the individual process identifiers. Therefore, when the context is clear, we omit the valuation pids of a state.

We consider the *transitions* of states induced by the program. A transition $t^{\{i\}}$ pertains to some process P_i . It transfers process P_i from control location l_1 to l_2 . In general, the application of $t^{\{i\}}$ is guarded by some condition `cond` (`cond` might

be just `true`). At some state $s \in State$, when the i^{th} component of $\text{pc}(s)$, namely $\text{pc}(s)[i]$, equals 1_1 , we say that $t^{\{i\}}$ can be *scheduled* at s . And when the valuation $\text{val}(s);\text{pids}$ satisfies the guard `cond`, denoted by $\text{val}(s);\text{pids} \models \text{cond}$, we say that $t^{\{i\}}$ is *enabled* at s . Furthermore, we call the enabling condition of $t^{\{i\}}$ the formula: $(\text{pc}(s)[i] == 1_1) \wedge \text{cond}$. For each state s , let $Scheduled(s)$ and $Enabled(s)$ denote the set of transitions which respectively can be scheduled at s and are enabled at s . Without further ado, we assume that the effect of applying an enabled transition $t^{\{i\}}$ on a state s to arrive at state s' is well-understood. This is denoted as $s \xrightarrow{t^{\{i\}}} s'$.

Again consider in Figure 3 with two processes P_1 and P_2 with variables $\text{id}_1 = 1$ and $\text{id}_2 = 2$ respectively. In the system, it is specified parametrically that each process awaits for `x == id`. In P_1 , this is interpreted as `await(x == id1)` while P_2 , this is interpreted as `await(x == id2)`. Each process has 2 transitions: the first transfers it from control location $\langle 0 \rangle$ to $\langle 1 \rangle$, whereas the second transfers it from control location $\langle 1 \rangle$ to $\langle 2 \rangle$. Initially we have $x = 1$, i.e. the initial state s_0 is $\langle \langle 0, 0 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle$. We note that at s_0 , both $t_1^{\{1\}}$ and $t_1^{\{2\}}$ can be scheduled. However, among them, only $t_1^{\{1\}}$ is enabled. By taking transition $t_1^{\{1\}}$, P_1 moves from control location $\langle 0 \rangle$ to $\langle 1 \rangle$, and the whole system moves from state $\langle \langle 0, 0 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle$ to state $\langle \langle 1, 0 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle$. We note that here the transition $t_1^{\{2\}}$ is still disabled. From now on, let us omit the valuation of process identifiers. The whole system then takes the transition $t_2^{\{1\}}$ and moves from state $\langle \langle 1, 0 \rangle; x = 1 \rangle$ to state $\langle \langle 2, 0 \rangle; x = 2 \rangle$. Now, $t_1^{\{2\}}$ becomes enabled. Subsequently, the system takes $t_1^{\{2\}}$ and $t_2^{\{2\}}$ to move to state $\langle \langle 2, 1 \rangle; x = 1 \rangle$ and finally to state $\langle \langle 2, 2 \rangle; x = 3 \rangle$.

Definition 1 (Safety). *We say a given concurrent system is safe wrt. a safety property ψ if $\forall s \in State \bullet s$ is reachable from s_0 implies $s \models \psi$.*

2.1 Symmetry

Given an n -process system, let $\mathcal{I} = [1 \dots n]$ denote its *indices*, to be thought of as process identifiers. We write $Sym \mathcal{I}$ to denote the set of all permutations π on index set \mathcal{I} . Let Id be the identity permutation and π^{-1} the inverse of π .

For an indexed object b , such as a program point, a variable, a transition, valuation of program variables, or a formula, whose definition depends on \mathcal{I} , we can define the notion of permutation π acting on b , by simultaneously replacing each occurrence of index $i \in \mathcal{I}$ by $\pi(i)$ in b to get the result of $\pi(b)$.

Example 1. Consider the system in Figure 3(b). Let the permutation π swap the two indices ($1 \mapsto 2, 2 \mapsto 1$). Applying π to the valuation $x = 1$ gives us $\pi(x = 1) \equiv x = 1$, as x is a shared variable. Applying π to the formula $x = id_1 \wedge id_1 = 1$ gives us $\pi(x = id_1 \wedge id_1 = 1) \equiv (x = id_2 \wedge id_2 = 1)$. On the other hand, applying π to the transition $t_1^{\{1\}} \equiv \text{await}(x = id_1)$ will result in $\pi(t_1^{\{1\}}) \equiv t_1^{\{2\}} \equiv \text{await}(x = id_2)$.

Definition 2. For $\pi \in \text{Sym } \mathcal{I}$ and state $s \in \text{State}$, $s \equiv \langle \text{pc}(s); \text{val}(s); \text{pids} \rangle$, the application of π on s is defined as $\langle \pi(\text{pc}(s)); \pi(\text{val}(s)); \pi(\text{pids}) \rangle$,

In other words, permutations do not affect the valuation of process identifiers.

Example 2. Consider again the system in Figure 3(b). Assume the π is the permutation swapping the 2 indices ($1 \mapsto 2, 2 \mapsto 1$). We then can have $\pi(\langle \langle 1, 0 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle) \equiv \langle \langle 0, 1 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle$. Please note that while π has no effect on shared variable x and valuation of process identifiers id_1, id_2 , it does permute the local program points.

Definition 3. For $\pi \in \text{Sym } \mathcal{I}$, a safety property ψ is said to be symmetric wrt. π if $\psi \equiv \pi(\psi)$.

We next present a traditional notion of symmetry.

Definition 4 (Strong Symmetry). For $\pi \in \text{Sym } \mathcal{I}$, and a safety property ψ , for $s, s' \in \text{State}$, we say that s is strongly π -similar to s' wrt. ψ , denoted by $s \xrightarrow{\pi, \psi} s'$ if ψ is symmetric wrt. π and the following conditions hold:

- $\pi(s) = s'$
- for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \xrightarrow{\pi, \psi} d'$
- for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \xrightarrow{\pi, \psi} d'$.

One of the strengths of this paper is to allow symmetry by *disregarding* the values of the program variables.

Definition 5 (Weak Symmetry). For $\pi \in \text{Sym } \mathcal{I}$, and a safety property ψ , for $s, s' \in \text{State}$, we say that s is weakly π -similar to s' wrt. ψ , denoted by $s \xrightarrow{\pi, \psi} s'$ if ψ is symmetric wrt. π and the following conditions hold:

- $\pi(\text{pc}(s)) = \text{pc}(s')$
- $s \models \psi$ iff $s' \models \pi(\psi)$
- for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \xrightarrow{\pi, \psi} d'$
- for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \xrightarrow{\pi, \psi} d'$.

We note here that, from now on, unless otherwise mentioned, symmetry means *weak* symmetry while π -similar means *weakly* π -similar. Also, it trivially follows that if s is π -similar to s' then s' is π^{-1} -similar to s . Consequently, if s is symmetric with s' , then s' is symmetric with s too.

2.2 State Interpolation

State-based interpolation was first described in [10] for finite transition systems. The essential idea was to prune the search space of symbolic execution, informally described as follows. Symbolic execution is usually depicted as a tree rooted at the initial state s_0 and for each state s_i therein, the descendants are just the states obtainable by extending s_i with an enabled transition. Consider one

particular feasible path represented in the tree: $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \dots \xrightarrow{t_m} s_m$. We adapt the usual notion of *program point*, characterizing a point in the reachability tree in terms of all the remaining possible transitions. Now, this particular path is *safe* wrt. to safety property ψ if for all i , $0 \leq i \leq m$, we have $s_i \models \psi$. A (state) interpolant at program point j , $0 \leq j \leq m$ is simply a set of states S_j containing s_j such that for any state $s'_j \in S_j$, $s'_j \xrightarrow{t_{j+1}} s'_{j+1} \xrightarrow{t_{j+2}} s'_{j+2} \dots \xrightarrow{t_m} s'_m$, it is also the case that for all i , $j \leq i \leq m$, we have $s'_i \models \psi$. This interpolant was constructed at point j due to the one path. Consider now all paths from s_0 and with prefix t_1, \dots, t_{j-1} . Compute each of their interpolants. Finally, we say that the interpolant for the subtree of paths just considered is simply the intersection of all the individual interpolants. This notion of interpolant for a subtree provides a notion of *subsumption* because we can now prune a subtree in case the root of this subtree are within the interpolant computed for some previously encountered subtree of the same program point.

Definition 6 (Safe Root). Let s_i be a state which is reachable from the initial state s_0 , we say that s_i is a safe root, denoted by $\Delta(s_i)$, if for all state s'_i reachable from s_i , s_i is safe.

Definition 7 (State Coverage). Let s_i and s_j be two states which are reachable from the initial state s_0 such that $\text{pc}(s_i) \equiv \text{pc}(s_j)$. We say that s_i covers s_j , denoted by $s_i \succeq s_j$, if $\Delta(s_i)$ implies $\Delta(s_j)$.

During the traversal of the reachability tree, if we detect that $s_i \succeq s_j$ while s_i has been proved to be a safe root, the traversal of the subtree rooted at s_j can be avoided. We thus reduce the search space.

Definition 8 (Sound Interpolant). Let pp be a global program point. We say a formula ϕ is a sound interpolant for pp if for all state s reachable from the initial state s_0 , $\text{pc}(s) \equiv \text{pp} \wedge s \models \phi$ implies that s is a safe root.

In practice, in order to determine state coverage, during the exploration of subtree rooted at s_i we compute an interpolant of $\text{pc}(s_i)$, denoted as $\text{SI}(\text{pc}(s_i), \psi)$, where ψ is the target safety property. Note that trivially, we should have $s_i \models \text{SI}(\text{pc}(s_i), \psi)$. We assume that this condition is always ensured by any implementation of our state-based interpolation. Furthermore, $\text{SI}(\text{pc}(s_i), \psi)$ ensures that $\forall s_j \in \text{State} \bullet \text{pc}(s_j) \equiv \text{pc}(s_i) \wedge s_j \models \text{SI}(s_i, \psi)$, then for all $t \in \text{Scheduled}(s_i)$ ¹, the two following conditions must be satisfied:

- if t was disabled at s_i , it also must be disabled at s_j
- if t was enabled at s_j (by the above condition, it must be enabled at s_i) and $s_j \xrightarrow{t} s'_j$ and $s_i \xrightarrow{t} s'_i$, then s'_i must cover s'_j .

This observation enables us to determine the coverage relation as the form of backward learning in a recursive manner. Our symmetry reduction algorithm presented in Section 4 will implement this idea of state interpolation.

¹ Since $\text{pc}(s_j) \equiv \text{pc}(s_i)$, we have $\text{Scheduled}(s_j) \equiv \text{Scheduled}(s_i)$.

3 Motivating Examples

Figure 4 shows a parameterized system and its 2-process concretization. The shared array x contains 2 elements, initially 0. For convenience, we assume that array index starts from 1. Process 1 assigns id_1 (whose value is 1) to $x[1]$ while process 2 assigns id_2 (2) to $x[2]$.

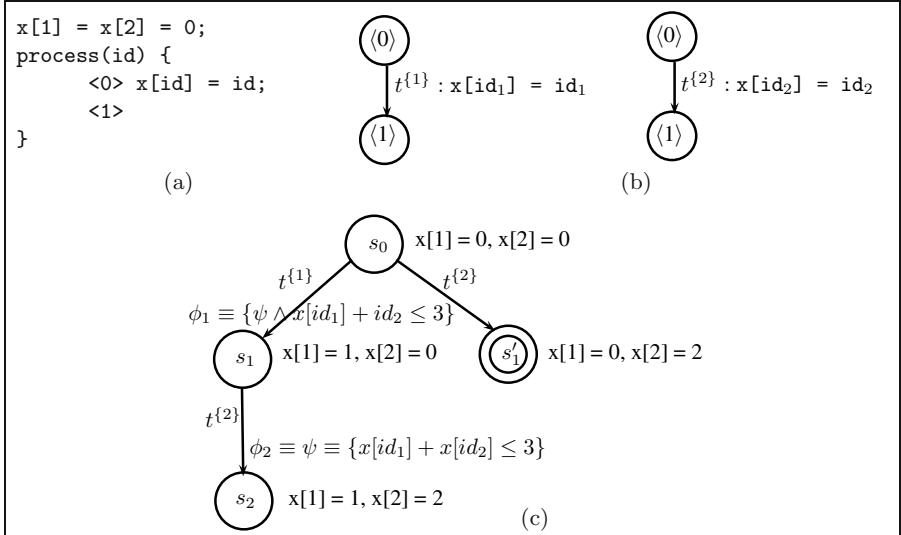
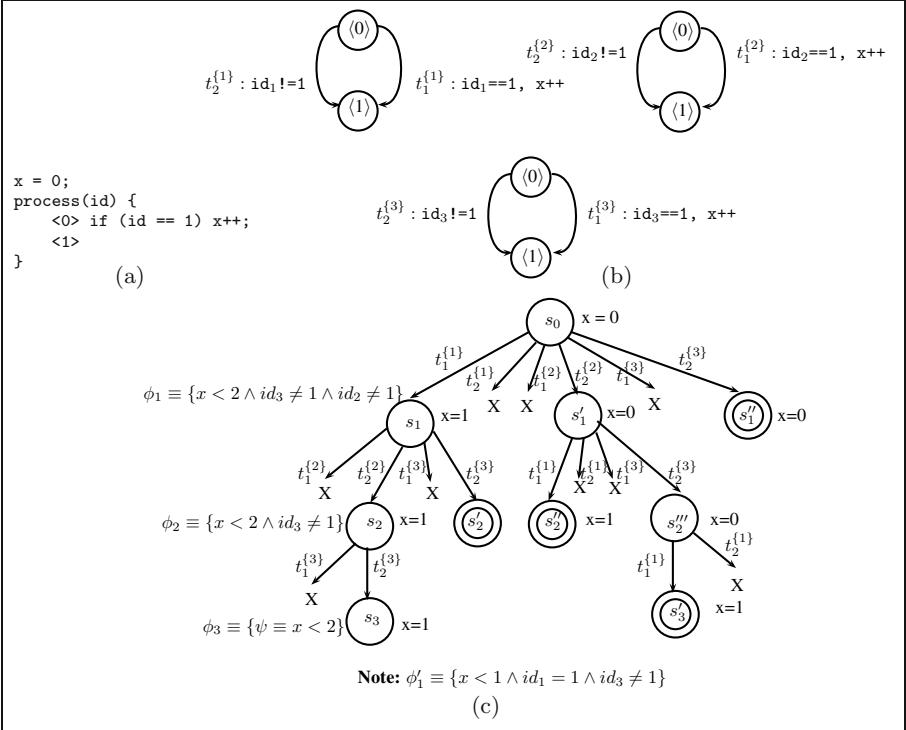


Fig. 4. (a) An example (b) Its 2-process concretization (c) Traversed tree

Consider the safety property $\psi \equiv x[1] + x[2] \leq 3$, interpreted as $\psi \equiv x[id_1] + x[id_2] \leq 3$. The reachability tree explored is in Figure 4(c). Circles are used to denote states, while double-boundary circles denote subsumed/pruned states.

From the initial state $s_0 \equiv \langle \langle 0, 0 \rangle; x[1] = 0, x[2] = 0; id_1 = 1, id_2 = 2 \rangle$ process 1 progresses first and moves the system to the state $s_1 \equiv \langle \langle 1, 0 \rangle; x[1] = 1, x[2] = 0; id_1 = 1, id_2 = 2 \rangle$. From s_1 , process 2 now progresses and moves the system to the state $s_2 \equiv \langle \langle 1, 1 \rangle; x[1] = 1, x[2] = 2; id_1 = 1, id_2 = 2 \rangle$. Note that s_0 , s_1 , and s_2 are all safe wrt. ψ . As there is no transition emanating from s_2 , the interpolant for s_2 is computed as $\phi_2 \equiv \psi \equiv x[id_1] + x[id_2] \leq 3$. The pair $\langle \langle 1, 1 \rangle; \phi_2 \rangle$ is memoized. The interpolant for s_1 can be computed as a conjunction of two formulas. One concerns the safety of s_1 itself, and the other concerns the safety of the successor state from $t^{(2)}$. In other words, we can have $\phi_1 \equiv \psi \wedge \text{pre}(x[id_2] = id_2; \psi)$, where $\text{pre}(t; \phi)$ denotes a precondition wrt. to the program transition t and the postcondition ϕ . Consequently, we can have $\phi_1 \equiv \psi \wedge x[id_1] + id_2 \leq 3$. The pair $\langle \langle 1, 0 \rangle; \phi_1 \rangle$ is memoized.

Now we arrive at state $s'_1 \equiv \langle \langle 0, 1 \rangle; x[1] = 0, x[2] = 2; id_1 = 1, id_2 = 2 \rangle$. This is indeed a symmetric image of state s_1 which we have explored and proved to be safe before. Here, we discover the permutation π to transform the program

**Fig. 5.** (a) An example (b) Its 3-process concretization (c) Traversed tree

point $\langle 1, 0 \rangle$ to program point $\langle 1, 0 \rangle$. Clearly π simply swaps the two indices. We also observe that the safety property ψ is symmetric wrt. this π , i.e. $\pi(\psi) \equiv \psi$ (ψ is *invariant* wrt. π). In the next step, we check whether $\text{val}(s'_1)$ conjoined with pids implies the *transformed interpolant* $\pi(\phi_1)$. We have $\pi(\phi_1) \equiv \pi(x[id_1] + x[id_2] \leq 3 \wedge x[id_1] + id_2 \leq 3 \equiv x[id_2] + x[id_1] \leq 3 \wedge x[id_2] + id_1 \leq 3)$. As $\text{val}(s'_1); \text{pids} \models x[id_2] + x[id_1] \leq 3 \wedge x[id_2] + id_1 \leq 3$, we do not need to explore s'_1 any further. In other words, the subtree rooted at s'_1 is pruned.

Another example is Figure 5. We are interested in safety property $\psi \equiv x < 2$. As x is a shared variable, ψ is symmetric wrt. all possible permutations.

The reachability tree is depicted in Figure 5(c). From the initials state s_0 we arrive at states s_1 , s_2 , and s_3 , where:

$$\begin{aligned}
s_0 &\equiv \langle \langle 0, 0, 0 \rangle; x = 0; id_1 = 1, id_2 = 2, id_3 = 3 \rangle \\
s_1 &\equiv \langle \langle 1, 0, 0 \rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3 \rangle \\
s_2 &\equiv \langle \langle 1, 1, 0 \rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3 \rangle \\
s_3 &\equiv \langle \langle 1, 1, 1 \rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3 \rangle.
\end{aligned}$$

At s_3 we compute its interpolant $\phi_3 \equiv \psi \equiv x < 2$. In a similar manner as before, we compute the interpolant for s_2 , which is $\phi_2 \equiv x < 2 \wedge id_3 \neq 1$. When we are at state $s'_2 \equiv \langle \langle 1, 0, 1 \rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3 \rangle$, we look for a

permutation π_1 such that $\pi_1(\langle 1, 1, 0 \rangle) = \langle 1, 0, 1 \rangle$. Clearly we can have π_1 as the permutation which fixes the first index and swaps the last 2 indices. Moreover, $\text{val}(s'_2); \text{pids} \equiv x = 1; id_1 = 1, id_2 = 2, id_3 = 3 \models \pi_1(\phi_2) \equiv x < 2 \wedge id_2 \neq 1$. Therefore, s'_2 is pruned.

Similarly, the interpolant for s_1 is computed as $\phi_1 \equiv x < 2 \wedge id_2 \neq 1 \wedge id_3 \neq 1$. When at state $s'_1 \equiv \langle \langle 0, 1, 0 \rangle; x = 0; id_1 = 1, id_2 = 2, id_3 = 3 \rangle$, we look for a permutation π_2 such that $\pi_2(\langle 1, 0, 0 \rangle) = \langle 0, 1, 0 \rangle$. Clearly we can have π_2 as the permutation which fixes the third index and swaps the first two indices. However, $\text{val}(s'_1); \text{pids} \equiv x = 0; id_1 = 1, id_2 = 2, id_3 = 3 \not\models \pi_2(\phi_1) \equiv x < 2 \wedge id_1 \neq 1 \wedge id_3 \neq 1$. Thus the subtree rooted at s'_1 cannot be pruned and it requires further exploration. After s'_1 has been traversed, the interpolant for s'_1 is computed as $\phi'_1 \equiv x < 1 \wedge id_1 = 1 \wedge id_3 \neq 1$. Next we arrive at $s''_1 \equiv \langle \langle 0, 0, 1 \rangle; x = 0; id_1 = 1, id_2 = 2, id_3 = 3 \rangle$. We can find a permutation π_3 which fixes the first index and swaps the last 2 indices ($\pi_3 \equiv \pi_1$). We have $\pi_3(\langle 0, 1, 0 \rangle) = \langle 0, 0, 1 \rangle$. Also $\text{val}(s''_1); \text{pids} \equiv x = 0; id_1 = 1, id_2 = 2, id_3 = 3 \models \pi_3(\phi'_1) \equiv x < 1 \wedge id_1 = 1 \wedge id_2 \neq 1$. As a result, we can avoid considering the subtree rooted at s''_1 .

In the two examples above, we have shown how the concept of interpolation can help capture the shape of a subtree. More importantly, computed interpolants can be transformed in order to detect the symmetry as well as the non-symmetry (mainly due to the use of `id`) between candidate subtrees.

4 Symmetry Reduction Algorithm

Our algorithm, presented in Figure 6, naturally performs a depth first search of the interleaving tree. It assumes the safety property to be known as ψ . Initially, we explore the initial state s_0 with an empty *history*. During the search process, the function `Explore` will be recursively called. Note that termination is ensured due to the finite domain of discourse.

Base Cases: The first base case is when the current state does not conform to the safety property ψ (line 2). We then immediately report an error and terminate. The second base case applies when the current state (subtree) has a symmetric image (subtree) which has already been traversed and proved to be safe before (line 3). We have well exemplified such scenarios in previous sections.

The third base case requires some elaboration. Using the history h , we detect a cycle (line 4). Specifically, there is a cyclic path θ from s back to s . We note this down and return true. Later, after the descendants of s have been traversed, we require a fix-point computation of the interpolant for s , as shown in line 17-18. The function `FIX-POINT` computes an invariant interpolant for s , wrt. the initial value ϕ and the set of cyclic paths Θ . Essentially, this function involves computing, for each cyclic path, a *path invariant*. Such a computation is performed backwards, using a previously computed invariant at the bottom of the cyclic path, and then extracting a new invariant for s . Then each computed path invariant is fed into other paths in order to compute a new invariant. The process terminates at a fix-point. Termination is guaranteed because of monotonicity of the path invariant computation and the fact that there are only finitely many possible invariants (the state s itself is an invariant). Finally, the interpolant for

```

⟨1⟩ Initially : Explore( $s_0, \emptyset$ )
function Explore( $s, h$ )
⟨2⟩ if  $s \not\models \psi$  then Report Error and TERMINATE
⟨3⟩ if  $\exists \pi \bullet \pi(\psi) \equiv \psi \wedge \exists pp \bullet pc(s) \equiv \pi(pp) \wedge \exists \phi \bullet memoed(pp, \phi) \wedge s \models \pi(\phi)$  then
    return  $\pi(\phi)$ 
⟨4⟩ if  $s \in h$                                 /* We hit a cycle */
⟨5⟩ let  $\theta$  be the cyclic path
⟨6⟩ Assert(Cyclic( $s, \theta$ ))
⟨7⟩ return true                               /* Initial value for fix-point computation */
else
⟨8⟩    $h \leftarrow h \cup \{s\}$ 
endif
⟨9⟩  $\phi \leftarrow \psi$ 
⟨10⟩ foreach  $t \in Scheduled(s)$  do
    if  $t \in Enabled(s)$ 
    ⟨12⟩    $s' \leftarrow$  successor of  $s$  after  $t$           /* Execute t */
    ⟨13⟩    $\phi' \leftarrow$  Explore( $s', h$ )
    ⟨14⟩    $\phi \leftarrow \phi \wedge pre(t; \phi')$ 
    else
    ⟨15⟩        $\phi \leftarrow \phi \wedge pre(t; false)$ 
    endif
⟨16⟩ endfor
⟨17⟩ let  $\Theta$  be  $\{\theta \mid Cyclic(s, \theta)\}$ 
⟨18⟩ if  $\Theta \neq \emptyset$  then  $\phi \leftarrow$  FIX-POINT( $s, \Theta, \phi$ )
    /*  $s$  is a looping point, so we ensure  $\phi$  is an invariant along the paths  $\Theta$  */
⟨19⟩ Retractall(Cyclic( $s, \theta$ ))
⟨20⟩  $h \leftarrow h \setminus \{s\}$ 
⟨21⟩ memo(pc( $s$ ),  $\phi$ ) and return  $\phi$ 
end function

```

Fig. 6. Symmetry Reduction Algorithm (DFS)

each state appearing in these cyclic paths are now updated appropriately. This is in light of now having an invariant for all of them simultaneously.

We remark here that this fix-point task, though seemingly complicated, is in fact routine. We refer interested readers to [9] for more details regarding this matter. We also remark that for many concurrent protocols, where involved operations are mainly “set” and “re-set” operations, a fix-point is achieved just after one iteration.

Recursive Traversal and Computing the Interpolants: Our algorithm recursively explores the successors of the current state by the recursive call in line 13. The interpolant ϕ for the current state is computed as from line 9-18. As mentioned above, cyclic paths are handled in line 17-18. The operation $pre(t; \bar{\phi})$ denotes the precondition computation wrt. the program transition t and the postcondition $\bar{\phi}$. In practice, we implement this as an approximation of the weakest precondition computation [3].

Theorem 1 (Soundness). *Our symmetry reduction algorithm is sound.*

Here, by soundness, we mean that all pruning performed in line 3 will not affect the verification result.

Proof (Outline). Let the triple $\{\phi\} \langle\langle pc_1, pc_2, \dots, pc_n \rangle; P_1 || P_2 || \dots || P_n \rangle \{\psi\}$ denote the fact that ϕ is a *sound* interpolant for program point $\langle pc_1, pc_2, \dots, pc_n \rangle$ wrt. the safety property ψ and the concurrent system $P_1 || P_2 || \dots || P_n$. Due to space limit, we will not prove that our interpolant computation (line 9-18) is a sound computation. Instead, we refer interested readers to [10,9]. Let us assume that the soundness of that triple is witnessed by a proof \mathcal{P} . By consistently renaming \mathcal{P} with a renaming function $\pi \in Sym \mathcal{I}$, we can derive a new *sound* fact (i.e. a proof), which is:

$$\begin{aligned} \{\pi(\phi)\} \pi(\langle\langle pc_1, pc_2, \dots, pc_n \rangle; P_1 || P_2 || \dots || P_n \rangle) \{\pi(\psi)\} \equiv \\ \{\pi(\phi)\} \langle\langle pc_{\pi(1)}, pc_{\pi(2)}, \dots, pc_{\pi(n)} \rangle; P_{\pi(1)} || P_{\pi(2)} || \dots || P_{\pi(n)} \rangle \{\pi(\psi)\} \end{aligned}$$

Since P_1, P_2, \dots, P_n come from the same parameterized system and π is a bijection on \mathcal{I} , we have:

$$P_{\pi(1)} || P_{\pi(2)} || \dots || P_{\pi(n)} \equiv P_1 || P_2 || \dots || P_n$$

Therefore, $\{\pi(\phi)\} \langle\langle pc_{\pi(1)}, pc_{\pi(2)}, \dots, pc_{\pi(n)} \rangle; P_1 || P_2 || \dots || P_n \rangle \{\pi(\psi)\}$ must hold too. In the case that ψ is symmetric wrt. π , we have $\pi(\psi) \equiv \psi$. Thus $\pi(\phi)$ is a *sound* interpolant for program point $\langle pc_{\pi(1)}, pc_{\pi(2)}, \dots, pc_{\pi(n)} \rangle$ wrt. the same safety property ψ and the same concurrent system $P_1 || P_2 || \dots || P_n$. As a result, the use of interpolant $\pi(\phi)$ at line 3 in our algorithm is *sound*. \square

Definition 9 (Symmetry Preserving Precondition Computation). *Given a parametrically defined n-process system and a safety property ψ , the precondition computation pre used in our algorithm is said to be symmetry preserving if for all $\pi \in Sym \mathcal{I}$, for all transition t and all postcondition ϕ • $pre(t; \phi) \equiv pre(\pi(t); \pi(\phi))$.*

This property means that our precondition computation is consistent wrt. to renaming operation. In other words, the implementation of pre is *independent* of the naming of variables contained in its inputs. A reasonable implementation of pre can easily ensure this.

Definition 10 (Monotonic Precondition Computation). *Given a parametrically defined n-process system and a safety property ψ , the precondition computation pre used in our algorithm is said to be monotonic if for all transition t and all postconditions ϕ_1, ϕ_2 • $\phi_1 \rightarrow \phi_2$ implies $pre(t; \phi_1) \rightarrow pre(t; \phi_2)$.*

We emphasize here that the weakest precondition computation [8] does possess the monotonicity property. As is well-known, computing the weakest precondition in all the cases is very expensive. However, in practice (and in particular in the experiments we have performed), we often observe this monotonicity property with the implementation of our precondition computation. Incidentally, some possible implementations for this operation are discussed in [11,10,9,1].

Definition 11 (Completeness). *In proving a parametrically defined n-process system with a global state space $State$ and a safety property ψ , an algorithm which*

traverses the reachability tree is said to be complete wrt. a symmetry relation \mathcal{R} iff for all $s, s' \in \text{State}$, $s \mathcal{R} s'$ implies that the algorithm will avoid traversing either the subtree rooted at s or the subtree rooted at s' .

We remark here that our definition of completeness does not concern with the power of an algorithm in giving the answer to a safety verification question. This definition of completeness, however, is about the power of an algorithm in exploiting symmetry for search space reduction.

Theorem 2 (Completeness). *Our symmetry reduction algorithm is complete wrt. the weak symmetry relation if our operation pre is both monotonic and symmetry preserving.*

Proof (Outline). Assume that $s, s' \in \text{State}$ and s is weakly π -similar to s' . W.l.o.g. assume we encounter s first. If the subtree rooted at s is pruned (due to subsumption), the theorem trivially holds. The theorem also trivially holds if s is *not* a safe root. Now we consider that the subtree rooted as s is proved to be safe and the returned interpolant is ϕ . We will prove by structural induction on this interpolated subtree that s' will indeed be pruned, i.e. $s' \models \pi(\phi)$.

For simplicity of the proof, we will prove for loop-free programs only. In other words, we ignore our loop handling mechanism (line 4-7,17-18). Note that our theorem still holds for the general case. However, to prove this, we will require another induction on our fix-point computation in line 18.

For the base case that ϕ is ψ (when there is no schedulable transition from s) due to the definition of weak symmetry relation, there is no schedulable transition from s' and $s' \models \pi(\psi)$. Therefore, traversing the subtree rooted at s' is avoided.

As the induction hypothesis, assume now that the theorem holds for all the descendants of state s . Let assume that $\phi \equiv \psi \wedge \phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_k \wedge \phi_{k+1} \wedge \dots \wedge \phi_m$, where $\phi_1 \dots \phi_k$ are the interpolants contributed by enabled transitions in s and $\phi_{k+1} \dots \phi_m$ are the interpolants contributed by schedulable but disabled transitions in s (line 14 and 15). Now assume the contrary that $s' \not\models \pi(\phi)$. We will show that this would lead to a contradiction. Using the first condition of weak symmetry relation, obviously $s' \models \pi(\psi)$. As such, there must exist some $1 \leq j \leq m$ such that $s' \not\models \pi(\phi_j)$. There are two possible cases: (1) ϕ_j is contributed by an enabled transition; (2) ϕ_j contributed by a disabled, but can be scheduled, transition.

Let us consider case (1) first. Assume ϕ_j corresponds to transition $t \in \text{Enabled}(s)$ and $s \xrightarrow{t} d$. By definition we have $s' \xrightarrow{\pi(t)} d'$ and d is weakly π -similar to d' . Let ϕ_d be interpolant for the subtree rooted at d . By induction hypothesis, we have $d' \models \pi(\phi_d)$. Obviously, we have $s' \models \text{pre}(\pi(t); d')$, by monotonicity of pre , we deduce $s' \models \text{pre}(\pi(t); \pi(\phi_d))$. As pre is symmetry preserving, $s' \models \text{pre}(\pi(t); \pi(\phi_d)) \equiv \pi(\text{pre}(t; \phi_d)) \equiv \pi(\phi_j)$. Consequently we arrive at the fact that $s' \models \pi(\phi_j)$ which is a contradiction.

For case (2), by using the symmetry preserving property of pre and the fact that $\pi(\text{false}) \equiv \text{false}$, we also derive a contradiction. \square

5 Experimental Evaluation

We used a 3.2 GHz Intel processor and 2GB memory running Linux. Unless otherwise mentioned, timeout is set at 5 minutes, and ‘-’ indicates timeout. In this section, we benchmark our proposed approach, namely Complete Symmetry Reduction (CSR), against current state-of-the-arts.

Table 1. Experiments on Dining Philosophers

# Phil	CSR			RSR			NSR		
	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
3	68	29	0.02	67	27	0.02	191	79	0.06
4	230	134	0.09	328	184	0.13	1246	702	0.81
5	662	446	0.28	1509	981	0.71	7517	4893	4.93
6	1778	1304	0.85	7356	5216	4.18	43580	30908	34.53
7	4584	3552	2.55	35079	26335	28.83	-	-	-
8	11526	9281	7.54	-	-	-	-	-	-
9	28287	23432	22.6	-	-	-	-	-	-
10	67920	57504	58.07	-	-	-	-	-	-
11	159738	137609	226.86	-	-	-	-	-	-

Our first example is the classic *dining philosophers* problem. As commonly known, it exhibits *rotational* symmetry. However, and more importantly, we exploit far more symmetry than that. More specifically, at *any* program point, rotational symmetry is applicable. Nevertheless, for certain program points, when some transitions have been taken, the system exhibits more symmetry than just rotational symmetry. With this benchmark, we demonstrate the power of our complete symmetry reduction (CSR) algorithm. Here, we verify a *tight* safety property that ‘no more than *half* the philosophers can eat simultaneously’.

Table 1 presents three variants: Complete Symmetry Reduction (CSR), Rotational Symmetry Reduction (RSR), and No Symmetry Reduction (NSR). The number of *stored states* is the difference between the number of visited states (Visited column) and subsumed states (Subsumed column). Note that although RSR achieves linear reduction compared to NSR, it does not scale well. CSR significantly outperforms RSR and NSR in all the instances.

Next consider the *Reader-Writer Protocol* from [14,15]. Here we highlight the aspect of *search space size* as compared to top-down techniques, of which the most recent implementation of Lazy Symmetry Reduction [15] is chosen as a representative [2]. Table 2 shows that although lazy symmetry reduction has aggressively compressed the state space (which now grows roughly in linear complexity), the running time is still *exponential*. In other words, the number of abstract states is not representative of the search space. In contrast, the running time of our method is significantly better. In the instance of 8 readers and 4 writers, we extended the timeout for [15] to finish; and it takes almost 1 day.

² We receive this implementation from the authors of [15].

Table 2. Experiments on Reader-Writer Protocol

		Complete Symmetry Reduction			Lazy Symmetry Reduction		
# Readers	# Writers	Visited	Subsumed	T(s)	Abstract States	T(s)	
2	1	35	20	0.01	9	0.01	
4	2	226	175	0.19	41	0.10	
6	3	779	658	0.93	79	67.80	
8	4	1987	1750	3.23	165	81969.00	
10	5	4231	3820	9.21	—	—	

Table 3. Experiments on sum-of-ids Example

		Complete Symmetry Reduction			SPIN-NSR		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)	
10	57	45	0.02	6146	4097	0.03	
20	212	190	0.04	11534338	9437185	69.70	
40	822	780	0.37	—	—	—	
60	1832	1770	1.91	—	—	—	
80	3242	3160	7.62	—	—	—	
100	5052	4950	22.09	—	—	—	

Next we experiment with the ‘sum-of-ids’ example mentioned earlier. To the best of our knowledge, there is no symmetry reduction algorithm which can detect and exploit symmetry here. Table 3 shows we have significant symmetry reduction. In term of memory (stored states), we enjoy linear complexity. For reference, we also report the running time of this example, without symmetry reduction, using SPIN 5.1.4 [13].

Table 4. Experiments on Bakery Algorithm

		Complete Symmetry Reduction			SI		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)	
3	65	31	0.10	265	125	0.43	
4	182	105	0.46	1925	1089	5.89	
5	505	325	2.26	14236	9067	74.92	
6	1423	983	11.10	—	—	—	

In the fourth and last example, we depart slightly from our finite domain to allow infinite domain variables and loops. We choose the well-known Bakery Algorithm to perform the experiments, and we use the well-known abstraction of using an inequality to describe each pair of counters to close the loops. Again, as far as we are aware of, there has been no symmetry reduction algorithm which can detect and exploit symmetry for this example. Table 4 shows the significant improvements due to our symmetry reduction, compared to just symbolic execution with interpolation, denoted as SI.

6 Conclusion

We presented a method of symmetry reduction for searching the interleaving space of a concurrent system of transitions in pursuit of a safety property. The class of systems considered, by virtue of being defined parametrically, is completely general; the individual processes may be at any level of similarity to each other. We then enhanced a general method of symbolic execution with interpolation for traditional safety verification of transition systems, in order to deal with symmetric states. We then defined a notion of weak symmetry, one that allows for more symmetry than the stronger notion that is used in the literature. Finally, we showed that our method, when employed with an interpolation algorithm which is monotonic, can exploit weak symmetry completely.

References

1. Chu, D.H., Jaffar, J.: Symbolic simulation on complicated loops for WCET path analysis. In: EMSOFT, pp. 319–328 (2011)
2. Clarke, E.M., Filkorn, T., Jha, S.: Exploiting Symmetry in Temporal Logic Model Checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 450–462. Springer, Heidelberg (1993)
3. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 453–457 (1975)
4. Emerson, E.A., Havlicek, J.W., Tretler, R.J.: Virtual symmetry reduction. In: Logic in Computer Science, pp. 121–131 (2000)
5. Emerson, E.A., Sistla, A.P.: Model Checking and Symmetry. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993)
6. Emerson, E.A., Sistla, A.P.: Utilizing symmetry when model-checking under fairness assumptions. *ACM Trans. Program. Lang. Syst.* 19(4), 617–638 (1997)
7. Emerson, E.A., Tretler, R.J.: From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 142–157. Springer, Heidelberg (1999)
8. Ip, C.N., Dill, D.L.: Better verification through symmetry. *Form. Methods Syst. Des.* 9(1/2), 41–75 (1996)
9. Jaffar, J., Navas, J.A., Santosa, A.E.: Unbounded Symbolic Execution for Program Verification. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 396–411. Springer, Heidelberg (2012)
10. Jaffar, J., Santosa, A.E., Voicu, R.: An Interpolation Method for CLP Traversal. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 454–469. Springer, Heidelberg (2009)
11. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
12. Sistla, A.P., Godefroid, P.: Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.* 26(4), 702–734 (2004)
13. SPIN model checker, <http://spinroot.com>
14. Wahl, T.: Adaptive Symmetry Reduction. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 393–405. Springer, Heidelberg (2007)
15. Wahl, T., D’Silva, V.: A lazy approach to symmetry reduction. *Form. Asp. Comput.* 22, 713–733 (2010)

Synthesizing Number Transformations from Input-Output Examples

Rishabh Singh^{1,*} and Sumit Gulwani²

¹ MIT CSAIL, Cambridge, MA, USA

² Microsoft Research, Redmond, WA, USA

Abstract. Numbers are one of the most widely used data type in programming languages. Number transformations like formatting and rounding present a challenge even for experienced programmers as they find it difficult to remember different number format strings supported by different programming languages. These transformations present an even bigger challenge for end-users of spreadsheet systems like Microsoft Excel where providing such custom format strings is beyond their expertise. In our extensive case study of help forums of many programming languages and Excel, we found that both programmers and end-users struggle with these number transformations, but are able to easily express their intent using input-output examples.

In this paper, we present a framework that can learn such number transformations from very few input-output examples. We first describe an expressive number transformation language that can model these transformations, and then present an inductive synthesis algorithm that can learn all expressions in this language that are consistent with a given set of examples. We also present a ranking scheme of these expressions that enables efficient learning of the desired transformation from very few examples. By combining our inductive synthesis algorithm for number transformations with an inductive synthesis algorithm for syntactic string transformations, we are able to obtain an inductive synthesis algorithm for manipulating data types that have numbers as a constituent sub-type such as date, unit, and time. We have implemented our algorithms as an Excel add-in and have evaluated it successfully over several benchmarks obtained from the help forums and the Excel product team.

1 Introduction

Numbers represent one of the most widely used data type in programming languages. Number transformations like formatting and rounding present a challenge even for experienced programmers. First, the custom number format strings for formatting numbers are complex and take some time to get accustomed to, and second, different programming languages support different format strings, which makes it difficult for programmers to remember each variant.

* Work done during an internship at Microsoft Research.

Number transformations present an even bigger challenge for end-users: the large class of users who do not have a programming background but want to create small, *often one-off*, applications to support business functions [4]. Spreadsheet systems like Microsoft Excel support a finite set of commonly used number formats and also let users write their own custom formats using a number formatting language similar to that of .NET. This hard-coded set of number formats is often insufficient for the user’s needs and providing custom number formats is typically beyond their expertise. This leads them to solicit help on various online help forums, where experts typically respond with the desired formulas (or scripts) after few rounds of interaction, which spans over a few days.

In an extensive case study of help forums of many programming languages and Excel, we found that even though both programmers and end-users struggled while performing these transformations, they were able to easily express their intent using input-output examples. In fact, in some cases the initial English description of the task provided by the users on forums was inaccurate and only after they provided a few input-output examples, the forum experts could provide the desired code snippet.

In this paper, we present a framework to learn number formatting and rounding transformations from a given set of input-output examples. We first describe a domain-specific language for performing number transformations and an inductive synthesis algorithm to learn the set of all expressions that are consistent with the user-provided examples. The key idea in the algorithm is to use the interval abstract domain [2] to represent a large collection of consistent format expressions symbolically, which also allows for efficient intersection, enumeration, and execution of these expressions. We also present a ranking mechanism to rank these expressions that enables efficient learning of the desired transformation from very few examples.

We then combine the number transformation language with a syntactic string transformation language [6] and present an inductive synthesis algorithm for the combined language. The combined language lets us model transformations on strings that represent data types consisting of number as a constituent subtype such as date, unit, time, and currency. The key idea in the algorithm is to succinctly represent an exponential number of consistent expressions in the combined language using a **Dag** data structure, which is similar to the **BDD** [1] representation of Boolean formulas. The **Dag** data structure consists of program expressions on the edges (as opposed to Boolean values on BDD edges). Similar to the **BDDs**, our data structure does not create a quadratic blowup after intersection in practice.

We have implemented our algorithms both as a stand-alone binary and as an Excel add-in. We have evaluated it successfully on over 50 representative benchmark problems obtained from help forums and the Excel product team.

This paper makes the following key contributions:

- We develop an expressive number transformation language for performing number formatting and rounding transformations, and an inductive synthesis algorithm for learning expressions in it.

- We combine the number transformation language with a syntactic string transformation language to manipulate richer data types.
- We describe an experimental prototype of our system with an attractive user interface that is ready to be deployed. We present the evaluation of our system over a large number of benchmark examples.

2 Motivating Examples

We motivate our framework with the help of a few examples taken from Excel help forums.

Example 1 (Date Manipulation). An Excel user stated that, as an unavoidable outcome of data extraction from a software package, she ended up with a series of dates in the input column v_1 as shown in the table. She wanted to convert them into a consistent date format as shown in the output column such that both month and day in the date are of two digits.

Input v_1	Output
1112011	01/11/2011
12012011	12/01/2011
1252010	01/25/2010
11152011	11/15/2011

It turns out that no Excel date format string matches the string in input column v_1 . The user struggled to format the date as desired and posted the problem on a help forum. After a few rounds of interactions (in which the user provided additional examples), the user managed to obtain the following formula for performing the transformation:

```
TEXT(IF(LEN(A1)=8,DATE(RIGHT(A1,4),MID(A1,3,2),LEFT(A1,2)),  
DATE(RIGHT(A1,4),MID(A1,2,2),LEFT(A1,1))),"mm/dd/yyyy")
```

In our tool, the user has to provide only the first two input-output examples from which the tool learns the desired transformation, and executes the synthesized transformation on the remaining strings in the input column to produce the corresponding outputs (shown in bold font for emphasis).

We now briefly describe some of the challenges involved in learning this transformation. We first require a way to extract different substrings of the input date for extracting the day, month, and year parts of the date, which can be performed using the syntactic string transformation language [6]. We then require a number transformation language that can map 1 to 01, i.e. format a number to two digits. Consider the first input-output example 1112011 \rightarrow 01/11/2011. The first two characters in the output string can be obtained by extracting 1 from the input string from any of the five locations where it occurs, and formatting it to 01 using a number format expression. Alternatively, the first 0 in the output string can also be a constant string or can be obtained from the 3rd last character in the input. All these different choices for each substring of the output string leads to an exponential number of choices for the complete transformation. We use an efficient data structure for succinctly representing such exponential number of consistent expressions in polynomial space.

Example 2 (Duration Manipulation). An Excel user wanted to convert the “raw data” in the input column to the lower range of the corresponding “30-min interval” as shown in the output column. An expert responded by providing the following macro, which is quite unreadable and error-prone.

Input v_1	Output
0d 5h 26m	5:00
0d 4h 57m	4:30
0d 4h 27m	4:00
0d 3h 57m	3:30

```
FLOOR(TIME(MID(C1,FIND(" ",C1)+1,FIND("h",C1)- FIND(" ",C1)-1)+0,
MID(C1,FIND("h",C1)+2,FIND("m",C1)-FIND("h",C1)-2)+0,0)*24,0.5)/24
```

Our tool learns the desired transformation using only the first two examples. In this case, we first need to be able to extract the hour and minute components of the duration in input column v_1 , and then perform a rounding operation on the minute part of the input to round it to the lower 30-min interval.

3 Overview of the Synthesis Approach

In this section, we define the formalism that we use in the paper for developing inductive synthesizers [8].

Domain-Specific Language: We develop a domain-specific language L that is expressive enough to capture the desired tasks and, at the same time, is concise for enabling efficient learning from examples.

Data Structure for Representing a Set of Expressions: The number of expressions that are consistent with a given input-output example can potentially be very large. We, therefore, develop an efficient data structure D that can succinctly represent a large number of expressions in L .

Synthesis Algorithm: The synthesis algorithm **Synthesize** consists of the following two procedures:

- **GenerateStr:** The **GenerateStr** procedure learns the set of all expressions in the language L (represented using the data structure D) that are consistent with a given input-output example (σ_i, s_i) . An input state σ holds values for m string variables v_1, \dots, v_m (denoting m input columns in a spreadsheet).
- **Intersect:** The **Intersect** procedure intersects two sets of expressions to compute the common set of expressions.

The synthesis algorithm **Synthesize** takes as input a set of input-output examples and generates a set of expressions in L that are consistent with them. It uses **GenerateStr** procedure to generate a set of expressions for each individual input-output example and then uses the **Intersect** procedure to intersect the corresponding sets to compute the common set of expressions.

```
Synthesize(( $\sigma_1, s_1), \dots, (\sigma_n, s_n))$ 
```

$$\begin{aligned} P &:= \text{GenerateStr}(\sigma_1, s_1); \\ \text{for } i &= 2 \text{ to } n: \\ P' &:= \text{GenerateStr}(\sigma_i, s_i); \\ P &:= \text{Intersect}(P, P'); \\ \text{return } P; \end{aligned}$$

Ranking: Since there are typically a large number of consistent expressions for each input-output example, we rank them using the Occam’s razor principle that

states that smaller and simpler explanations are usually the correct ones. This enables users to provide only a few input-output examples for quick convergence to the desired transformation.

4 Number Transformations

In this section, we first describe the number transformation language L_n that can perform formatting and rounding transformations on numbers. We then describe an efficient data structure to succinctly represent a large number of expressions in L_n , and present an inductive synthesis algorithm to learn all expressions in the language that are consistent with a given set of input-output examples.

4.1 Number Transformation Language L_n

The syntax of the number transformation language L_n is shown in Figure 1(a). The top-level expression e_n of the language denotes a number formatting expression of one of the following forms:

- $\text{Dec}(u, \eta_1, f)$: formats the number u in decimal form (e.g. 1.23), where η_1 denotes the number format for the integer part of u ($\text{Int}(u)$), and f represents the optional format consisting of the decimal separator and the number format for the fractional part ($\text{Frac}(u)$).
- $\text{Exp}(u, \eta_1, f, \eta_2)$: formats the number u in exponential form (e.g. 1.23E+2). It consists of an additional number format η_2 as compared to the decimal format expression, which denotes the number format of the exponent digits of u .

$\begin{aligned} \text{Expr. } e_n &:= \text{Dec}(u, \eta_1, f) \\ &\quad \quad \text{Exp}(u, \eta_1, f, \eta_2) \\ &\quad \quad \text{Ord}(u) \\ &\quad \quad \text{Word}(u) \\ &\quad \quad u \\ \text{Dec. Fmt. } f &:= (\odot, \eta) \quad \quad \perp \\ \text{Number } u &:= v_i \\ &\quad \quad \text{Round}(v_i, r) \\ \text{Round Fmt. } r &:= (z, \delta, m) \\ \text{Mode } m &:= \downarrow \quad \quad \uparrow \quad \quad \updownarrow \\ \text{Num. Fmt. } \eta &:= (\alpha, \beta, \gamma) \end{aligned}$	$\begin{aligned} \llbracket \text{Dec}(u, \eta_1, f) \rrbracket \sigma &= \llbracket (\text{Int}(\llbracket u \rrbracket^R, \eta_1) \rrbracket^R \sigma * \llbracket f \rrbracket \sigma \\ \llbracket \text{Exp}(u, \eta_1, f, \eta_2) \rrbracket \sigma &= \llbracket (\text{Int}(\llbracket u \rrbracket^R, \eta_1) \rrbracket^R \sigma * \llbracket f \rrbracket \sigma * \llbracket (\text{E}(\llbracket u \rrbracket^R, \eta_2) \rrbracket^R \sigma \\ \llbracket \text{Ord}(u) \rrbracket \sigma &= \text{numToOrd}(\llbracket u \rrbracket \sigma) \\ \llbracket \text{Word}(u) \rrbracket \sigma &= \text{numToWord}(\llbracket u \rrbracket \sigma) \\ \llbracket (\odot, \eta) \rrbracket \sigma &= \llbracket \odot \rrbracket \sigma * \llbracket (\text{Frac}(\llbracket u \rrbracket), \eta) \rrbracket \sigma \\ \llbracket \perp \rrbracket \sigma &= \epsilon \\ \llbracket v_i \rrbracket \sigma &= \sigma(v_i) \\ \llbracket \text{Round}(v_i, r) \rrbracket \sigma &= \text{RoundNumber}(\sigma(v_i), z, \delta, m) \\ &\quad \text{where } r = (z, \delta, m) \\ \llbracket (d, \eta) \rrbracket \sigma &= \text{FormatDigits}(d, \alpha, \beta, \gamma) \\ &\quad \text{where } \eta = (\alpha, \beta, \gamma) \end{aligned}$
(a)	(b)

Fig. 1. The (a) syntax and (b) semantics of the number transformation language L_n . The variable v_i denotes an input number variable, z , δ , α , β , and γ are integer constants, and $*$ denotes the concatenation operation.

RoundNumber(n, z, δ, m)

```

1  $n' := \left\lceil \frac{n - z}{\delta} \right\rceil \times \delta + z;$ 
2 if ( $n = n'$ ) return  $n$ ;
3 if ( $m = \uparrow$ ) return  $n' + \delta$ ;
4 if ( $m = \downarrow$ ) return  $n'$ ;
5 if ( $m = \downarrow \wedge (n - n') \times 2 < \delta$ )
    return  $n'$ ;
6 if ( $m = \downarrow \wedge (n - n') \times 2 \geq \delta$ )
    return  $n' + \delta$ ;
(a)

```

FormatDigits(d, α, β, γ)

```

1 if ( $\text{len}(d) \geq \beta$ )
2     return significant( $d, \beta$ );
3 else if ( $\text{len}(d) \geq \alpha$ )
4     { $z := 0$ ;  $s := 0$ ;}
5 else { $s := \text{Min}(\gamma, \alpha - \text{len}(d))$ ;
6          $z := \alpha - \text{len}(d) - s$ ;}
7 return concat( $d, 0_z, 's$ );
(b)

```

Fig. 2. The functions (a) `RoundNumber` for rounding numbers and (b) `FormatDigits` for formatting a digit string

- $\text{Ord}(u)$: formats the number u in *ordinal* form, e.g. it formats the number 4 to its ordinal form 4th.
- $\text{Word}(u)$: formats the number u in *word* form, e.g. it formats the number 4 to its word form *four*.

The number u can either be an input number variable v_i or a number obtained after performing a rounding transformation on an input number. A rounding transformation $\text{Round}(v_i, z, \delta, m)$ performs the rounding of number present in v_i based on its rounding format (z, δ, m) , where z denotes the *zero* of the rounding interval, δ denotes the interval size of the rounding interval, and m denotes one of the rounding mode from the set of modes $\{\text{upper}(\uparrow), \text{lower}(\downarrow), \text{nearest}(\Downarrow)\}$.

We define a *digit string* d to be a sequence of digits with trailing whitespaces. A number format η of a digit string d is defined by a 3-tuple (α, β, γ) , where α denotes the minimum number of significant digits and trailing whitespaces of d in the output string, β denotes the maximum number of significant digits of d in the output string, and γ denotes the maximum number of trailing whitespaces in the output string. A number format, thus, maintains the invariant: $\gamma \leq \alpha \leq \beta$.

The semantics of language L_n is shown in Figure 2(b). A digit string d is formatted with a number format (α, β, γ) using the `FormatDigits` function shown in Figure 2(b). The `FormatDigits` function returns the first β digits of the digit string d (with appropriate rounding) if the length of d is greater than the maximum number of significant digits β to be printed. If the length of d is lesser than β but greater than the minimum number of significant digits α to be printed, it returns the digits itself. Finally, if the length of d is less than α , it appends the digit string with appropriate number of zeros (z) and whitespaces (s) as computed in Lines 5 and 6. The semantics of the rounding transformation is to perform the appropriate rounding of number denoted by v_i using the `RoundNumber` function shown in Figure 2(a). The function computes a number n' which lies on the number line defined by zero z with unit separation δ as shown in Figure 3. It returns the value n' or $(n' + \delta)$ based on the rounding mode m and the distance between n and n' as described in Figure 2(a).

The semantics of a decimal form formatting expression on a number u is to concatenate the reverse of the string obtained by formatting the reverse of

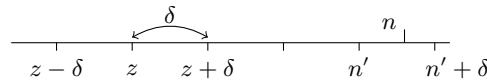


Fig. 3. The `RoundNumber` function rounding-off number n to n' or $n' + \delta$

integral part `Int`(u) with the string obtained from the decimal format f . Since the `FormatDigits` function adds only trailing zeros and whitespaces to format a digit string, the formatting of the integer part of u is performed on its reverse digit string and the resulting formatted string is reversed again before performing the concatenation. The semantics of decimal format f is to concatenate the decimal separator \odot with the string obtained by formatting the fractional part `Frac`(u). The semantics of exponential form formatting expression is similar to that of the decimal form formatting expression and the semantics of ordinal form and word form formatting expressions is to simply convert the number u into its corresponding ordinal form and word form respectively.

We now present some examples taken from various help forums that can be represented in the number transformation language L_n .

Example 3. A python programmer posted a query on the `StackOverflow` forum after struggling to print double values from an array of doubles (of different lengths) such that the decimal point for each value is aligned consistently across different columns. He posted an example of the desired formatting as shown on the right. He also wanted to print a single 0 after the decimal if the double value had no decimal part.

Input v_1	Output
3264.28	3264.28
53.5645	53.5645
235	235.0
5.23	5.23
345.213	345.213
3857.82	3857.82
536	536.0

The programmer started the post saying “*This should be easy*”. An expert replied that after a thorough investigation, he couldn’t find a way to perform this task without some post-processing. The expert provided the following python snippet that pads spaces to the left and zeros to the right of the decimal, and then removes trailing zeros:

```
ut0 = re.compile(r'(\d)0+$')
thelist = textwrap.dedent(
    '\n'.join(ut0.sub(r'\1', "%20f" % x) for x in a)).splitlines()
print '\n'.join(thelist)
```

This formatting transformation can be represented in L_n as `Dec`($v_1, \eta_1, (\cdot, \eta_2)$), where $\eta_1 \equiv (4, \infty, 4)$ and $\eta_2 \equiv (4, \infty, 3)$.

Example 4. This is an interesting post taken from a help forum where the user initially posted that she wanted to round numbers in an excel column to nearest 45 or 95, but the examples later showed that she actually wanted to round it to *upper* 45 or 95.

Input v_1	Output
11	45
32	45
46	95
1865	1895

Some of the solutions suggested by experts were:

```
=Min(Roundup(A1/45,0)*45,Roundup(A1/95,0)*95)
=CEILING(A1+5,50)-5
=A1-MOD(A1,100)+IF(MOD(A1,100)>45,95,45)
```

This rounding transformation can be expressed in our language as:

$$\text{Dec}(\text{Round}(v_1, (45, 50, \uparrow)), (0, \infty, 0), \perp).$$

4.2 Data Structure for a Set of Expressions in L_n

Figure 4 describes the syntax and semantics of the data structure for succinctly representing a set of expressions from language L_n . The expressions \tilde{e}_n are now associated with a set of numbers \tilde{u} and a set of number formats $\tilde{\eta}$. We represent the set of numbers obtained after performing rounding transformation in two ways: $\text{Round}(v_i, \tilde{r})$ and $\text{Round}(v_i, n_p)$, which we describe in more detail in section 4.3. The set of number formats $\tilde{\eta}$ are represented using a 3-tuple (i_1, i_2, i_3) , where i_1, i_2 and i_3 denote a set of values of α, β and γ respectively using an interval domain. This representation lets us represent $O(n^3)$ number of number format expressions in $O(1)$ space, where n denotes the length of each interval.

The semantics of evaluating the set of rounding transformations $\text{Round}(v_i, \tilde{r})$ is to return the set of results of performing rounding transformation on v_i for all rounding formats in the set \tilde{r} . The expression $\text{Round}(v_i, (n_1, n'_1))$ represents an infinite number of rounding transformations (as there exists an infinite number of rounding formats that conform to the rounding transformation $n_1 \rightarrow n'_1$). For evaluating this expression, we select one conforming rounding format with $z = 0, \delta = n'_1$ and an appropriate m as shown in the figure. The evaluation of a set of format strings $\tilde{\eta} = (i_1, i_2, i_3)$ on a digit string d returns a set of values, one for each possible combination of $\alpha \in i_1, \beta \in i_2$ and $\gamma \in i_3$. Similarly, we obtain a set of values from the evaluation of expression \tilde{e}_n .

4.3 Synthesis Algorithm

Procedure GenerateStr_n: The algorithm GenDFmt in Figure 5 takes as input two digit sequences d_1 and d_2 , and computes the set of all number formats $\tilde{\eta}$ that are consistent for formatting d_1 to d_2 . The algorithm first converts the digit sequence d_1 to its canonical form d'_1 by removing trailing zeros and whitespaces from d_1 . It then compares the lengths l_1 of d'_1 and l_2 of d_2 . If l_1 is greater than l_2 , then we can be sure that the digits got truncated and can therefore set the interval for i_2 (the maximum number of significant digits) to be $[l_2, l_2]$. The intervals for α and γ are set to $[0, l_2]$ because of the number format invariant. On the other hand if l_1 is smaller than l_2 , we can be sure that the least number of significant digits need to be l_2 , i.e. we can set the interval i_1 to be $[l_2, l_2]$. Also, we can set the interval i_2 to $[l_2, \infty]$ because of the number format invariant. For interval i_3 , we either set it to $[\xi, \xi]$ (when $l_2 - \xi \neq l_1$) or $[\xi, l_2]$ (when $l_2 - \xi = l_1$) where ξ denotes the number of trailing spaces in d_2 . In the former case, we can be sure about the exact number of trailing whitespaces to be printed.

$\begin{aligned} \bar{e}_n &:= \text{Dec}(\bar{u}, \tilde{\eta}_1, \tilde{f}) \\ &\quad \quad \text{Exp}(\bar{u}, \tilde{\eta}_1, \tilde{f}, \tilde{\eta}_2) \\ &\quad \quad \text{Ord}(\bar{u}) \\ &\quad \quad \text{Word}(\bar{u}) \\ &\quad \quad \bar{u} \\ \tilde{f} &:= (\odot, \tilde{\eta}) \quad \quad \perp \\ \bar{u} &:= v_i \\ &\quad \quad \text{Round}(v_i, \tilde{r}) \\ &\quad \quad \text{Round}(v_i, n_p) \\ \text{Pair } n_p &:= (n_1, n'_1) \\ \tilde{\eta} &:= (i_1, i_2, i_3) \\ \text{Interval } i &:= (l, h) \end{aligned}$	$\llbracket \text{Dec}(\bar{u}, \tilde{\eta}_1, \tilde{f}) \rrbracket = \{ \text{Dec}(u, \eta_1, f) \mid u \in \bar{u}, \eta_1 \in \tilde{\eta}_1, f \in \tilde{f} \}$ $\llbracket \text{Exp}(\bar{u}, \tilde{\eta}_1, \tilde{f}, \tilde{\eta}_2) \rrbracket = \{ \text{Exp}(u, \eta_1, f, \eta_2) \mid u \in \bar{u}, \eta_1 \in \tilde{\eta}_1,$ $f \in \tilde{f}, \eta_2 \in \tilde{\eta}_2 \}$ $\llbracket \text{Ord}(\bar{u}) \rrbracket = \{ \text{Ord}(u) \mid u \in \bar{u} \}$ $\llbracket \text{Word}(\bar{u}) \rrbracket = \{ \text{Word}(u) \mid u \in \bar{u} \}$ $\llbracket (\odot, \tilde{f}) \rrbracket = \{ (\odot, f) \mid f \in \tilde{f} \}$ $\llbracket \perp \rrbracket = \epsilon$ $\llbracket v_i \rrbracket = \{ v_i \}$ $\llbracket \text{Round}(v_i, \tilde{r}) \rrbracket = \{ \text{Round}(v_i, (z, \delta, m)) \mid (z, \delta, m) \in \tilde{r} \}$ $\llbracket \text{Round}(v_i, n_p) \rrbracket = \{ \text{Round}(v_i, (0, n'_1, m)) \mid n_p \equiv (n_1, n'_1),$ $\quad \text{if } (n_1 \leq n'_1) \text{ } m \equiv \uparrow \text{ else } m \equiv \downarrow \}$ $\llbracket (d, (i_1, i_2, i_3)) \rrbracket = \{ (d, \alpha, \beta, \gamma) \mid \alpha \in i_1, \beta \in i_2, \gamma \in i_3 \}$
(a)	(b)

Fig. 4. The (a) syntax and (b) semantics of a data structure for succinctly representing a set of expressions from language L_n .

The **GenerateStr_n** algorithm in Figure 5 learns the set of all expressions in L_n that are consistent with a given input-output example. The algorithm searches over all input variables v_i to find the inputs from which the output number n' can be obtained. It first converts the numbers $\sigma(v_i)$ and n' to their *canonical forms* n_c and n'_c respectively in Line 3. We define canonical form of a number to be its decimal value. If the two canonical forms n_c and n'_c are not equal, the algorithm tries to learn a rounding transformation such that n_c can be rounded to n'_c . We note that there is not enough information present in one input-output example pair to learn the exact rounding format as there exists an infinite family of such formats that are consistent. Therefore, we represent such rounding formats symbolically using the input-output example pair (n_c, n'_c) , which gets concretized by the **Intersect** method in Figure 6. The algorithm then normalizes the number $\sigma(u)$ with respect to n' using the **Normalize** method in Line 6 to obtain $n = (n_i, n_f, n_e)$ such that both n and n' are of the same form. For decimal and exponential forms, it learns a set of number formats $\tilde{\eta}$ for each of its constituent digit strings from the pairs $(n_i^R, n_i'^R)$, (n_f, n_f') , and $(n_e^R, n_e'^R)$ where n_i^R denotes the reverse of digit string n_i . As noted earlier, we need to learn the number format on the reversed digit strings for integer and exponential parts. For ordinal and word type numbers, it simply returns the expressions to compute ordinal and word forms of the corresponding input number respectively.

Procedure Intersect_n: The **Intersect_n** procedure for intersecting two sets of L_n expressions is described as a set of rules in Figure 6. The procedure computes the intersection of sets of expressions by recursively computing the intersection of their corresponding sets of sub-expressions. We describe below the four cases of

```

GenDFmt( $d_1$ : inp digits,  $d_2$ : out digits)
1  $d'_1 := \text{RemoveTrailingZerosSpaces}(d_1);$ 
2  $l_1 := \text{len}(d'_1); l_2 := \text{len}(d_2);$ 
3  $\xi := \text{numTrailingSpaces}(d_2);$ 
4 if ( $l_1 > l_2$ )
5   ( $i_1, i_2, i_3 := ([0, l_2], [l_2, l_2], [0, l_2]);$ 
6 else if ( $l_1 < l_2$ ) {
7    $i_1 := [l_2, l_2]; i_2 := [l_2, \infty];$ 
8     if ( $l_2 - \xi = l_1$ )  $i_3 := [\xi, l_2];$ 
9     else  $i_3 := [\xi, \xi];$ 
10 else ( $i_1, i_2, i_3 := ([0, l_2], [l_2, \infty], [0, l_2]);$ 
11 return  $\tilde{\eta}(i_1, i_2, i_3);$ 

Normalize( $n$ : inp number,  $n'$ : out number)
 $n_1 = n = (n_i, n_f, n_e);$ 
if( $\text{Type}(n) = \text{ExpNum} \wedge \text{Type}(n') \neq \text{ExpNum}$ )
   $n_1 := n \times 10^{n_e};$ 
if( $\text{Type}(n) \neq \text{ExpNum} \wedge \text{Type}(n') = \text{ExpNum}$ )
   $\{n' = (n'_i, n'_f, n'_e); n_1 := n / 10^{n_e};\}$ 
return  $n_1;$ 

GenerateStrn( $\sigma$ : inp state,  $n'$ : out number)
1  $S_n := \emptyset;$ 
2 foreach input variable  $v_i$ :
3    $n_c = \text{Canonical}(\sigma(v_i)); n'_c = \text{Canonical}(n');$ 
4   if ( $n_c \neq n'_c$ )  $u := \text{Round}(v_i, (n_c, n'_c));$ 
5   else  $u := v_i;$ 
6    $(n_i, n_f, n_e) := \text{Normalize}(\sigma(u), n');$ 
7   match  $n'$  with
8      $\text{DecNum}(n'_i, n'_f, \odot) \rightarrow$ 
9        $\tilde{\eta}_1 := \text{GenDFmt}(n_i^R, n_i'^R);$ 
10      if ( $\odot = \epsilon$ )  $S_n := S_n \cup \text{Dec}(u, \eta_1, \perp);$ 
11      else  $\{\tilde{\eta}_2 := \text{GenDFmt}(n_f, n_f');$ 
12         $S_n := S_n \cup \text{Dec}(u, \tilde{\eta}_1, \odot, \tilde{\eta}_2);\}$ 
13    $\text{ExpNum}(n'_i, n'_f, n'_e, \odot) \rightarrow$ 
14      $\tilde{\eta}_1 := \text{GenDFmt}(n_e^R, n_e'^R);$ 
15      $\tilde{\eta}_3 := \text{GenDFmt}(n_e^R, n_e'^R);$ 
16     if ( $\odot = \epsilon$ )  $S_n := S_n \cup \text{Exp}(u, \tilde{\eta}_1, \perp, \tilde{\eta}_3);$ 
17     else  $\{\tilde{\eta}_2 := \text{GenDFmt}(n_f, n_f');$ 
18        $S_n := S_n \cup \text{Exp}(u, \tilde{\eta}_1, \odot, \tilde{\eta}_2, \tilde{\eta}_3);\}$ 
19    $\text{OrdNum}(n'_i) \rightarrow$ 
20      $S_n := S_n \cup \text{Ord}(u);$ 
21    $\text{WordNum}(n'_i) \rightarrow$ 
22      $S_n := S_n \cup \text{Word}(u);$ 
23 return  $S_n;$ 

```

Fig. 5. The GenerateStr_n procedure for generating the set of all expressions in language L_n that are consistent with the given set of input-output examples

intersecting rounding transformation expressions. The first case is of intersecting a finite rounding format set \tilde{r} with another finite set \tilde{r}' . The other two cases intersect a finite set \tilde{r} with an input-output pair n_p , which is performed by selecting a subset of the finite set of rounding formats that are consistent with the pair n_p . The final case of intersecting two input-output pairs to obtain a finite set of rounding formats is performed using the IntersectPair algorithm shown in Figure 7

Consider the example of rounding numbers to nearest 45 or 95 for which we have the following two examples: $32 \rightarrow 45$ and $81 \rightarrow 95$. Our goal is to learn the rounding format (z, δ, m) that can perform the desired rounding transformation. We represent the infinite family of formats that satisfy the rounding constraint for each example as individual pairs $(32, 45)$ and $(81, 95)$ respectively. When we intersect these pairs, we can assign z to be 45 without loss of generality. We then compute all divisors $\tilde{\delta}$ of $95 - 45 = 50$. With the constraint that $\delta \geq (\text{Max}(45 - 32, 95 - 81) = 14)$, we finally arrive at the set $\tilde{\delta} = \{25, 50\}$. The rounding modes m are appropriately learned as shown in Figure 7. For decimal numbers, we compute the divisors by first scaling them appropriately and then re-scaling them back for learning the rounding formats.

```

IntersectPair(( $n_1, n'_1$ ), ( $n_2, n'_2$ ))
 $z := n'_1;$ 
 $\tilde{\delta} := \text{Divisors}(\|n'_2 - n'_1\|);$ 
 $S := \emptyset;$ 
foreach  $\delta \in \tilde{\delta}$ :
  if ( $\delta \geq \text{Max}(\|n_1 - n'_1\|, \|n_2 - n'_2\|))$ 
    if ( $2 \times \text{Max}(\|n_1 - n'_1\|, \|n_2 - n'_2\|) \leq \delta$ )
       $S := S \cup (z, \delta, \uparrow);$ 
    if ( $n_1 > n'_1 \wedge n_2 > n'_2$ )
       $S := S \cup (z, \delta, \downarrow);$ 
    if ( $n_1 < n'_1 \wedge n_2 < n'_2$ )
       $S := S \cup (z, \delta, \uparrow);$ 
  return  $S;$ 

```

Fig. 7. Intersection of Round expressions

$$\begin{aligned}
\text{Intersect}_n(\text{Dec}(\tilde{u}, \tilde{\eta}_1, \tilde{f}), \text{Dec}(\tilde{u}', \tilde{\eta}'_1, \tilde{f}')) &= \text{Dec}(\text{Intersect}_n(\tilde{u}, \tilde{u}'), \text{Intersect}_n(\tilde{\eta}_1, \tilde{\eta}'_1), \\
&\quad \text{Intersect}_n(\tilde{f}, \tilde{f}')) \\
\text{Intersect}_n(\text{Exp}(\tilde{u}, \tilde{\eta}_1, \tilde{f}, \tilde{\eta}_2), \text{Exp}(\tilde{u}, \tilde{\eta}'_1, \tilde{f}', \tilde{\eta}'_2)) &= \text{Exp}(\text{Intersect}_n(\tilde{u}, \tilde{u}'), \text{Intersect}_n(\tilde{\eta}_1, \tilde{\eta}'_1), \\
&\quad \text{Intersect}_n(\tilde{f}, \tilde{f}'), \text{Intersect}_n(\tilde{\eta}_2, \tilde{\eta}'_2)) \\
\text{Intersect}_n(\text{Ord}(\tilde{u}), \text{Ord}(\tilde{u}')) &= \text{Ord}(\text{Intersect}_n(\tilde{u}, \tilde{u}')) \\
\text{Intersect}_n(\text{Word}(\tilde{u}), \text{Word}(\tilde{u}')) &= \text{Word}(\text{Intersect}_n(\tilde{u}, \tilde{u}')) \\
\text{Intersect}_n(v_i, v_i) &= v_i \\
\text{Intersect}_n((\odot, \tilde{\eta}), (\odot', \tilde{\eta}')) &= (\text{Intersect}_n(\odot, \odot'), \text{Intersect}_n(\tilde{\eta}, \tilde{\eta}')) \\
\text{Intersect}_n(\text{Round}(v_i, \tilde{r}), \text{Round}(v_i, \tilde{r}')) &= \text{Round}(v_i, \text{Intersect}_n(\tilde{r}, \tilde{r}')) \\
\text{Intersect}_n(\text{Round}(v_i, \tilde{r}), \text{Round}(v_i, n_p)) &= \text{Round}(v_i, \text{Intersect}_n(\tilde{r}, n_p)) \\
\text{Intersect}_n(\text{Round}(v_i, n_p), \text{Round}(v_i, \tilde{r})) &= \text{Round}(v_i, \text{Intersect}_n(n_p, \tilde{r})) \\
\text{Intersect}_n(\text{Round}(v_i, n_p), \text{Round}(v_i, n'_p)) &= \text{Round}(v_i, \text{IntersectPair}(n_p, n'_p)) \\
\text{Intersect}_n((i_1, i_2, i_3), (i'_1, i'_2, i'_3)) &= (\text{Intersect}_n(i_1, i'_1), \text{Intersect}_n(i_2, i'_2), \\
&\quad \text{Intersect}_n(i_3, i'_3)) \\
\text{Intersect}_n((l, h), (l', h')) &= (\text{Max}(l, l'), \text{Min}(h, h'))
\end{aligned}$$

Fig. 6. The Intersect_n function for intersecting sets of expressions from language L_n . The Intersect_n function returns ϕ in all other case not covered above.

In our data structure, we do not store all divisors explicitly as this set might become too large for big numbers. We observe that we only need to store the greatest and least divisors amongst them, and then we can intersect two such sets efficiently by computing the gcd of the two corresponding greatest divisors and the lcm of the two corresponding least divisors.

Ranking: We rank higher the lower value for α in the interval i_1 (to prefer lesser trailing zeros and whitespaces), the higher value of β in i_2 (to minimize un-necessary number truncation), the lower value of γ in i_3 (to prefer trailing zeros more than trailing whitespaces), and the greatest divisor in the set of divisors $\tilde{\delta}$ of the rounding format (to minimize the length of rounding intervals). We rank expressions consisting of rounding transformations lower than the ones that consist of only number formatting expressions.

Theorem 1 (Correctness of Learning Algorithm for L_n).

- (a) The procedure GenerateStr_n is sound and complete. The complexity of GenerateStr_n is $O(|s|)$, where $|s|$ denotes the length of the output string.
- (b) The procedure Intersect_n is sound and complete.

Example 5. Figure 8 shows a range of number formatting transformations and presents the format strings that are required to be provided in Excel, .NET, Python and C, as well as the format expressions that are synthesized by our algorithm. An N.A. entry denotes that the corresponding formatting task cannot be done in the corresponding language.

Formatting of Doubles				
Input String	Output String	Excel/C# Format String	Python/C Format String	Synthesized format
123.4567 123.4	123.46 123.40	#.00	.2f	$\eta_1 \equiv ([0, 3], [3, \infty], [0, 3])$ $\eta_2 \equiv ([2, 2], [2, 2], [0, 0])$
123.4567 123.4	123.46 123.4	#.##	N.A.	$\eta_1 \equiv ([0, 3], [3, \infty], [0, 3])$ $\eta_2 \equiv ([0, 1], [2, 2], [0, 1])$
123.4567 3.4	123.46 03.40	00.00	05.2f	$\eta_1 \equiv ([2, 2], [3, \infty], [0, 0])$ $\eta_2 \equiv ([2, 2], [2, 2], [0, 0])$
123.4567 3.4	123.46 03.4	00.##	N.A.	$\eta_1 \equiv ([2, 2], [3, \infty], [0, 0])$ $\eta_2 \equiv ([0, 1], [2, 2], [0, 1])$
9723.00 0.823	9.723E+03 8.23E-01	#.#### E 00	N.A.	$\eta_1 \equiv ([0, 1], [1, \infty], [0, 1])$ $\eta_2 \equiv ([0, 3], [3, \infty], [0, 3])$ $\eta_3 \equiv ([2, 2], [2, \infty], [0, 0])$
243 12	00243 00012	00000	05d	$\eta_1 \equiv ([5, 5], [5, \infty], [0, 0])$
1.2 18	1.2_ 18._	#.??	N.A.	$\eta_1 \equiv ([0, 1], [2, \infty], [0, 1])$ $\eta_2 \equiv ([2, 2], [2, \infty], [2, 2])$
1.2 18	--1.2-- _18._	???????	N.A.	$\eta_1 \equiv ([3, 3], [3, \infty], [2, 3])$ $\eta_2 \equiv ([3, 3], [3, \infty], [3, 3])$
1.2 18	--1.20-- _18.00_-	????.00?	N.A.	$\eta_1 \equiv ([3, 3], [3, \infty], [2, 3])$ $\eta_2 \equiv ([3, 3], [3, \infty], [1, 1])$

Fig. 8. We compare the custom number format strings required to perform formatting of doubles in Excel/C# and Python/C languages. An N.A. entry in a format string denotes that the corresponding formatting is not possible using format strings only. The last column presents the corresponding L_n expressions (- denotes whitespaces).

5 Combining Number Transformations with Syntactic String Transformations

In this section, we present the combination of number transformation language L_n with the syntactic string transformation language L_s [6] to obtain the combined language L_c , which can model transformations on strings that contain numbers as substrings. We first present a brief background description of the syntactic string transformation language and then present the combined language L_c . We also present an inductive synthesis algorithm for L_c obtained by combining the inductive synthesis algorithms for L_n and L_s respectively.

Syntactic String Transformation Language L_s (Background) Gulwani [6] introduced an expression language for performing syntactic string transformations. We reproduce here a small subset of (the rules of) that language and call it L_s (with e_s being the top-level symbol) as shown in Figure 9. The formal semantics of L_s can be found in [6]. For completeness, we briefly describe some key aspects of this language. The top-level expression e_s is either an atomic expression f or is obtained by concatenating atomic expressions f_1, \dots, f_n using

$$\begin{aligned}
e_s := & \text{Concatenate}(f_1, \dots, f_n) \mid f \\
\text{Atomic expr } f := & \text{ConstStr}(s) \mid v_i \mid \text{SubStr}(v_i, p_1, p_2) \\
\text{Position } p := & k \mid \text{pos}(r_1, r_2, c) \\
\text{Integer expr } c := & k \mid k_1 w + k_2 \\
\text{Regular expr } r := & \epsilon \mid T \mid \text{TokenSeq}(T_1, \dots, T_n)
\end{aligned}$$

Fig. 9. The syntax of syntactic string transformation language L_s

the **Concatenate** constructor. Each atomic expression f can either be a constant string $\text{ConstStr}(s)$, an input string variable v_i , or a substring of some input string v_i . The substring expression $\text{SubStr}(v_i, p_1, p_2)$ is defined partly by two *position expressions* p_1 and p_2 , each of which implicitly refers to the (subject) string v_i and must evaluate to a position within the string v_i . (A string with ℓ characters has $\ell + 1$ positions, numbered from 0 to ℓ starting from left.) $\text{SubStr}(v_i, p_1, p_2)$ is the substring of string v_i in between positions p_1 and p_2 . A position expression represented by a non-negative constant k denotes the k^{th} position in the string. For a negative constant k , it denotes the $(\ell + 1 + k)^{th}$ position in the string, where $\ell = \text{Length}(s)$. $\text{pos}(r_1, r_2, c)$ is another position expression, where r_1 and r_2 are regular expressions and integer expression c evaluates to a non-zero integer. $\text{pos}(r_1, r_2, c)$ evaluates to a position t in the subject string s such that r_1 matches some suffix of $s[0 : t]$, and r_2 matches some prefix of $s[t : \ell]$, where $\ell = \text{Length}(s)$. Furthermore, if c is positive (negative), then t is the $|c|^{th}$ such match starting from the left side (right side). We use the expression $s[t_1 : t_2]$ to denote the substring of s between positions t_1 and t_2 . We use the notation $\text{SubStr2}(v_i, r, c)$ as an abbreviation to denote the c^{th} occurrence of regular expression r in v_i , i.e., $\text{SubStr}(v_i, \text{pos}(\epsilon, r, c), \text{pos}(r, \epsilon, c))$.

A regular expression r is either ϵ (which matches the empty string, and therefore can match at any position of any string), a token T , or a token sequence $\text{TokenSeq}(T_1, \dots, T_n)$. The tokens T range over a finite extensible set and typically correspond to character classes and special characters. For example, tokens **CapitalTok**, **NumTok**, and **WordTok** match a nonempty sequence of uppercase alphabetic characters, numeric digits, and alphanumeric characters respectively.

A **Dag** based data structure is used to succinctly represent a set of L_s expressions. The **Dag** structure consists of a node corresponding to each position in the output string s , and a map W maps an edge between node i and node j to the set of all L_c expressions that can compute the substring $s[i..j]$. This representation enables sharing of common subexpressions amongst the set of expressions and represents an exponential number of expressions using polynomial space.

Example 6. An Excel user wanted to modify the delimiter in dates present in a column from “/” to “-”, and gave the following input-output example “08/15/2010” → “08-15-2010”. An expression in L_s that can perform this transformation is: $\text{Concatenate}(f_1, \text{ConstStr}("-"), f_2, \text{ConstStr}("-"), f_3)$, where $f_1 \equiv \text{SubStr2}(v_1, \text{NumTok}, 1)$, $f_2 \equiv \text{SubStr2}(v_1, \text{NumTok}, 2)$, and $f_3 \equiv \text{SubStr2}(v_1,$

`NumTok`, 3). This expression constructs the output string by concatenating the first, second, and third numbers of input string with constant strings “-”.

5.1 The Combination Language L_c

The grammar rules R_c for the combined language L_c are obtained by taking the union of the rules for the two languages R_n and R_s with the top-level rule e_s . The modified rules are shown in the figure on the right. The combined language consists of an additional expression rule g that corresponds to either some input column v_i or a substring of some input column. This expression g is then passed over to the number variable expression u for performing number transformations on it. This rule enables the combined language to perform number transformations on substrings of input strings. The top-level expression of the number language e_n is added to the atomic expr f of the string language. This enables number transformation expressions to be present on the `Dag` edges together with the syntactic string transformation expressions.

The transformation in Example 10 is represented in L_c as: `Concatenate(f1, ConstStr("/", f2, ConstStr("/", f3)), where f1 ≡ Dec(g1, (2, ∞, 0), ⊥), g1 ≡ SubStr(v1, 1, -7), f2 ≡ SubStr(v1, -7, -5), and f3 ≡ SubStr(v1, -5, -1).. The transformation in Example 12 is represented as: Concatenate(f1, ":" , f2), where f1 ≡ SubStr2(v1, NumTok, 2), f2 ≡ Dec(u1, (2, ∞, 0), ⊥), and u1 ≡ Round(SubStr2(v1, NumTok, 3), (0, 30, ↓)).`

5.2 Data Structure for Representing a Set of Expressions in L_c

Let \tilde{R}_n and \tilde{R}_s denote the set of grammar rules for the data structures that represent a set of expressions in L_n and L_s respectively. We obtain the grammar rules \tilde{R}_c for succinctly representing a set of expressions of L_c by taking the union of the two rule sets \tilde{R}_n and \tilde{R}_s with the updated rules as shown in Figure 10(a). The updated rules have expected semantics and can be defined as in Figure 10(b).

$\tilde{f} := \dots \tilde{e}_n$ $\tilde{u} := \tilde{g} \text{Round}(\tilde{g}, \tilde{r})$ $\tilde{g} := v_i \text{SubStr}(v_i, \tilde{p}_1, \tilde{p}_2)$
--

(a)

```

GenerateStrc(σ: Inp, s: Out)
   $\tilde{\eta} = \{0, \dots, \text{Length}(s)\};$ 
   $\eta^s = 0;$ 
   $\eta^t = \text{Length}(s);$ 
   $\tilde{\xi} = \{\langle i, j \rangle \mid 0 \leq i < j < \text{Length}(s)\};$ 
  foreach substring s[i..j] of s:
    W[⟨i, j⟩] = ConstStr(s[i..j])
       $\cup$  GenerateStrs(σ, s[i..j])
       $\cup$  GenerateStr'n(σ, s[i..j])
  return Dag( $\tilde{\eta}$ ,  $\eta^s$ ,  $\eta^t$ ,  $\tilde{\xi}$ , W);

```

(b)

Fig. 10. (a) The data structure and (b) the `GenerateStrc` procedure for L_c expressions

5.3 Synthesis Algorithm

Procedure GenerateStr_c :

We first make the following two modifications in the GenerateStr_n procedure to obtain $\text{GenerateStr}'_n$ procedure. The first modification is that we now search over all substrings of input string variables v_i instead of just v_i in Line 2 in Figure 5. This lets us model transformations where number transformations are required to be performed on substrings of input strings. The second modification is that we replace each occurrence of v_i by $\text{GenerateStr}_s(\sigma, v_i)$ inside the loop body. This lets us learn the syntactic string program to extract the corresponding substring from the input string variables. The GenerateStr_c procedure for the combined language is shown in the Figure 10(b). The procedure first creates a Dag of $(\text{Length}(s) + 1)$ number of nodes with start node $\eta^s = 0$ and target node $\eta^t = \text{Length}(s)$. The procedure iterates over all substrings $s[i..j]$ of the output string s , and adds a constant string expression, a set of substring expressions (GenerateStr_s) and a set of number transformation expressions ($\text{GenerateStr}'_n$) that can generate the substring $s[i..j]$ from the input state σ . These expressions are then added to a map $W[\langle i, j \rangle]$, where W maps each edge $\langle i, j \rangle$ of the dag to a set of expressions in L_c that can generate the corresponding substring $s[i..j]$.

Procedure Intersect_c : The rules for Intersect_c procedure for intersecting sets of expressions in L_c are obtained by taking the union of intersection rules of Intersect_n and Intersect_s procedures together with corresponding intersection rules for the updated and new rules.

Ranking: The ranking scheme of the combined language L_c is obtained by combining the ranking schemes of languages L_n and L_s . In addition, we prefer substring expressions corresponding to longer input substrings that can be formatted or rounded to obtain the output number string.

Theorem 2 (Correctness of Learning Algorithm for combined language).

- (a) *The procedure GenerateStr_c is sound and complete with complexity $O(|s|^3 l^2)$, where $|s|$ denotes the length of the output string and l denotes the length of the longest input string.*
- (b) *The procedure Intersect_c is sound and complete.*

6 Experiments

We have implemented our algorithms in C# as an add-in to the Microsoft Excel spreadsheet system. The user provides input-output examples using an Excel table with a set of input and output columns. Our tool learns the expressions in L_c for each output column separately and executes the learned set of expressions on the remaining entries in the input columns to generate their corresponding outputs. We have evaluated our implementation on over 50 benchmarks obtained from various help forums, mailing lists, books and the Excel product team. More details about the benchmark problems can be found in [22].

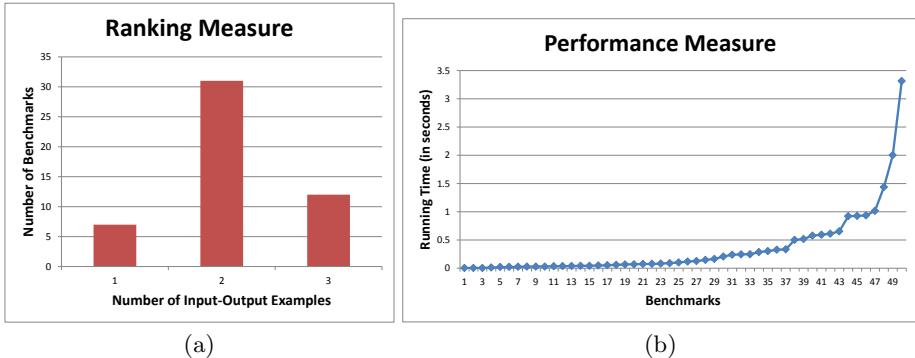


Fig. 11. (a) Number of examples required and (b) the running time of algorithm (in seconds) to learn the desired transformation

The results of our evaluation are shown in Figure 11. The experiments were run on an Intel Core-i7 1.87 Ghz CPU with 4GB of RAM. We evaluate our algorithm on the following two dimensions:

Ranking: Figure 11(a) shows the number of input-output examples required by our tool to learn the desired transformation. All benchmarks required at most 3 examples, with majority (76%) taking only 2 examples to learn the desired transformation. We ran this experiment in an automated counter-example guided manner such that given a set of input-output examples, we learned the transformations using a subset of the examples (training set). The tool iteratively added the failing test examples to the training set until the synthesized transformation conformed to all the remaining examples.

Performance: The running time of our tool on the benchmarks is shown in Figure 11(b). Our tool took at most 3.5 seconds each to learn the desired transformation for the benchmarks, with majority (94%) taking less than a second.

7 Related Work

The closest related work to ours is our previous work on synthesizing syntactic string transformations [6]. The algorithm presented in that work assumes strings to be a sequence of characters and can only perform concatenation of input substrings and constant strings to generate the desired output string. None of our benchmarks presented in this paper can be synthesized by that algorithm as it lacks reasoning about the semantics of numbers present in the input string.

There has been a lot of work in the HCI community for automating end-user tasks. Topes [20] system lets users create abstractions (called topes) for different data present in the spreadsheet. It involves defining constraints on the data to generate a context free grammar using a GUI and then this grammar is used to validate and reformat the data. There are several *programming by demonstration* [3] (PBD) systems that have been developed for data validation, cleaning

and formatting, which requires the user to specify a complete demonstration or trace visualization on a representative data instead of code. Some of such systems include Simultaneous Editing [18] for string manipulation, SMARTedit [17] for text manipulation and Wrangler [15] for table transformations. In contrast to these systems, our system is based on programming by example (PBE) – it requires the user to provide only the input and output examples without providing the intermediate configurations which renders our system more usable [16], although at the expense of making the learning problem harder. Our expression languages also learns more sophisticated transformations involving conditionals. The by-example interface [7] has also been developed for synthesizing bit-vector algorithms [14], spreadsheet macros [8] (including semantic string manipulation [21] and table layout manipulation [12]), and even some intelligent tutoring scenarios (such as geometry constructions [10] and algebra problems [23]).

Programming by example can be seen as an instantiation of the general program synthesis problem, where the provided input-output examples constitutes the specification. Program synthesis has been used recently to synthesize many classes of non-trivial algorithms, e.g. graph algorithms [13], bit-streaming programs [26][9], program inverses [27], interactive code snippets [11][19], and data-structures [24][25]. There are a range of techniques used in these systems including exhaustive search, constraint-based reasoning, probabilistic inference, type-based search, theorem proving and version-space algebra. A recent survey [5] explains them in more details. Lau et al. used the version-space algebra based technique for learning functions in a PBD setting [17], our system uses it for learning expressions in a PBE setting.

8 Conclusions

We have presented a number transformation language that can model number formatting and rounding transformations, and an inductive synthesis algorithm that can learn transformations in this language from a few input-output examples. We also showed how to combine our system for number transformations with the one for syntactic string transformations [6] to enable manipulation of data types that contain numbers as substrings (such as date and time). In addition to helping end-users who lack programming expertise, we believe that our system is also useful for programmers since it can provide a consistent number formatting interface across all programming languages.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers* 35(8), 677–691 (1986)
2. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
3. Cypher, A. (ed.): *Watch What I Do – Programming by Demonstration*. MIT Press (1993)

4. Gualtieri, M.: Deputize end-user developers to deliver business agility and reduce costs. Forrester Report for Application Development and Program Management Professionals (April 2009)
5. Gulwani, S.: Dimensions in program synthesis. In: PPDP (2010)
6. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL (2011)
7. Gulwani, S.: Synthesis from examples. In: WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings, vol. 10(2) (2012)
8. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Communications of the ACM (to appear, 2012)
9. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI (2011)
10. Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing geometry constructions. In: PLDI, pp. 50–61 (2011)
11. Gvero, T., Kuncak, V., Piskac, R.: Interactive Synthesis of Code Snippets. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 418–423. Springer, Heidelberg (2011)
12. Harris, W.R., Gulwani, S.: Spreadsheet table transformations from examples. In: PLDI, pp. 317–328 (2011)
13. Itzhaky, S., Gulwani, S., Immerman, N., Sagiv, M.: A simple inductive synthesis methodology and its applications. In: OOPSLA (2010)
14. Jha, S., Gulwani, S., Seshia, S., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (2010)
15. Kandel, S., Paepcke, A., Hellerstein, J., Heer, J.: Wrangler: Interactive visual specification of data transformation scripts. In: CHI (2011)
16. Lau, T.: Why PBD systems fail: Lessons learned for usable AI. In: CHI Workshop on Usable AI (2008)
17. Lau, T., Wolfman, S., Domingos, P., Weld, D.: Programming by demonstration using version space algebra. Machine Learning 53(1-2), 111–156 (2003)
18. Miller, R.C., Myers, B.A.: Interactive simultaneous editing of multiple text regions. In: USENIX Annual Technical Conference (2001)
19. Perelman, D., Gulwani, S., Ball, T., Grossman, D.: Type-directed completion of partial expressions. In: PLDI (2012)
20. Scaffidi, C., Myers, B.A., Shaw, M.: Topes: reusable abstractions for validating data. In: ICSE, pp. 1–10 (2008)
21. Singh, R., Gulwani, S.: Learning Semantic String Transformations from Examples. PVLDB 5(8), 740–751 (2012), http://vldb.org/pvldp/vol5/p740_rishabhsingh_vldp2012.pdf
22. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. Technical Report MSR-TR-2012-42 (April 2012)
23. Singh, R., Gulwani, S., Rajamani, S.: Automatically generating algebra problems. In: AAAI (to appear, 2012)
24. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: SIGSOFT FSE, pp. 289–299 (2011)
25. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: PLDI, pp. 136–148 (2008)
26. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294 (2005)
27. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. In: PLDI (2011)

Acacia+, a Tool for LTL Synthesis*

Aaron Bohy¹, Véronique Bruyère¹, Emmanuel Filiot²,
Naiyong Jin³, and Jean-François Raskin²

¹ Université de Mons (UMONS), Belgium

{aaron.bohy, veronique.bruyere}@umons.ac.be

² Université Libre de Bruxelles (ULB), Belgium

{jraskin, efiliot}@ulb.ac.be

³ Synopsys Inc.

nyjin@synopsys.com

Abstract. We present Acacia+, a tool for solving the LTL realizability and synthesis problems. We use recent approaches that reduce these problems to safety games, and can be solved efficiently by symbolic incremental algorithms based on antichains. The reduction to safety games offers very interesting properties in practice: the construction of compact solutions (when they exist) and a compositional approach for large conjunctions of LTL formulas.

Keywords: Church problem, LTL synthesis, antichains, safety games, Moore machines.

1 Introduction

LTL realizability and synthesis are central problems when reasoning about specifications for reactive systems. In the LTL realizability problem, the uncontrollable input signals are generated by the environment whereas the controllable output signals are generated by the system which tries to satisfy the specification against any behavior of the environment. The *LTL realizability problem* can be stated as a two-player game as follows. Let ϕ be an LTL formula over a set P partitioned into O (output signals controlled by Player O , the system) and I (input signals controlled by Player I , the environment). In the first round of the play, Player O starts by giving a subset $o_1 \subseteq O$ and Player I responds by giving a subset $i_1 \subseteq I$. Then the second round starts, Player O gives $o_2 \subseteq O$ and Player I responds by $i_2 \subseteq I$, and so on for an infinite number of rounds. The outcome of this interaction is the infinite word $w = (i_1 \cup o_1)(i_2 \cup o_2) \dots (i_k \cup o_k) \dots$ Player O wins the play if w satisfies ϕ , otherwise Player I wins. The realizability problem asks to decide whether Player O has a winning strategy to satisfy ϕ . The *LTL synthesis problem* asks to produce such a winning strategy when ϕ is realizable.

Due to their high worst-case complexities (2ExpTime-Complete), the LTL realizability and synthesis problems have been considered for a long time only of theoretical interest. Only recently, several progresses on algorithms and efficient data structures

* This work has been partly supported by the ESF project GASICS, the ARC project *Game Theory for the Automatic Synthesis of Computer Systems* and the ERC Starting Grant *inVEST*.

showed that they can also be solved in practice. It follows a renewed interest in these problems and a need for tools solving them. We participate to this research effort by providing a new tool, called **Acacia+**, that implements recent ideas that offer very interesting properties in practice: efficient symbolic incremental algorithms based on antichains, synthesis of small winning strategies (when they exist), compositional approach for large conjunctions of LTL formulas. This tool can be downloaded or simply used via a web interface. While its performances are better or similar to other existing tools, its main advantage is certainly the generation of *compact strategies* that are easily usable in practice. This aspect of **Acacia+** is very useful in several application scenarios, like synthesis of control code from high-level LTL specifications, debugging of unrealizable specifications by inspecting compact counter strategies, and generation of small deterministic automata from LTL formulas (when they exist).

2 Underlying Approach

LTL realizability and synthesis problems have been first studied in the seminal work [23]. The proposed solution is based on the costly Safra's procedure for the determinization of Rabin automata [4]. The LTL realizability problem is 2ExpTime-Complete and finite-memory strategies suffice to win the realizability game [52]. In [6], a so-called Safralless procedure avoids the determinization step by reducing the LTL realizability problem to Büchi games. It has been implemented in the tool **Lily** [78]. Another Safralless approach has been recently given in [9] for the distributed LTL synthesis problem. It is based on a novel emptiness-preserving translation from LTL to safety tree automata. In [10][11][12], a procedure for LTL synthesis problem is proposed and implemented in the tool **Unbeast**, based on the approach of [9] and symbolic game solving with BDDs.

Our tool **Acacia+** is based on several work by some authors of this paper [13][14]. In [13], a construction similar to [9] is proposed for LTL realizability and synthesis by a reduction to *safety games*. In this approach, the formula ϕ is first translated into an equivalent universal coBüchi word automaton, and then into an equivalent universal K -coBüchi automaton provided K is taken large enough (for which any infinite word w is accepted iff all runs labeled by w visit at most K accepting states). The latter automaton can be easily determinized with a variant of the classical subset construction, and the LTL synthesis problem is then solved *on the fly* as a safety game $G(\phi, K)$.

This approach offers very interesting properties in practice. (1) Checking the existence of a winning strategy for Player O in the game $G(\phi, K)$ can be done *incrementally* in the games $G(\phi, k)$, with $k = 0, 1, 2, \dots$ (when ϕ is realizable, $k \leq 5$ is often enough in practice). (2) When ϕ is unrealizable, by the determinacy of ω -regular games [15], Player I has a winning strategy for $\neg\phi$. Therefore checking the existence of a winning strategy for Player O is done incrementally in both games $G(\phi, k)$ and $G(\neg\phi, k)$, with $k = 0, 1, 2, \dots$ (3) The structure of $G(\phi, k)$ presents a partial-order on its states that can be used to represent compactly, with *antichains*, the set of *all* winning strategies. These three observations lead to an efficient antichain-based symbolic algorithm for the LTL realizability and synthesis problems, such that the antichain of the winning strategies for each player is obtained by a *backward fixpoint* computation from the safe configurations of $G(\phi, k)$ [13]. Moreover when ϕ is realizable, the computed antichain allows the

construction of a *compact* Moore machine representing a winning strategy for Player O (for Player I when ϕ is unrealizable). This algorithm is called *monolithic* in this paper.

In [13], the authors also propose two *compositional* algorithms for LTL formulas of the form $\phi = \phi_1 \wedge \cdots \wedge \phi_n$. The LTL realizability and synthesis problems are solved by first solving them separately for each conjunct ϕ_i , and then by composing the solutions according to the *parenthesizing* of ϕ . The first algorithm follows a compositional *backward* approach such that at each stage of the parenthesizing, the antichains W_i of the subformulae ϕ_i are computed backward and the antichain of the formula ϕ itself is also computed backward from the W_i 's. In this approach, *all* the winning strategies for ϕ (for a fixed k) are computed and compactly represented by the final antichain. This backward approach is optimized by considering relevant input signals only, called *critical signals* [14]. The second algorithm follows a compositional *forward* approach such that at each stage of parenthesizing, antichains are computed backward as explained before, except at the last stage where a forward algorithm seeks for *one* winning strategy by exploring the game arena on the fly in a forward fashion.

While the approaches detailed above ([13][14]) have been first implemented in a Perl prototype [16], we have reimplemented them from scratch in a new tool, **Acacia+**, now made available to the research community. This tool has been developed in Python and C, with emphasis on modularity, code efficiency, and usability. We hope that this will motivate other researchers to take up the available code and extend it. This new tool is detailed in Sect. 3 and typical scenarios of usage are presented in Sect. 4.

3 Tool Description

Programming Choices. **Acacia+** is written in Python and C. The C language is used for all the low level operations, while the orchestration is done with Python. The binding between these two languages is realized by the ctypes library of Python.

This separation presents two main advantages. (1) Due to the reduction to k -coBüchi automata and their determinization, we need to manipulate *counting functions* $Q \rightarrow \{-1, 0, \dots, k, k + 1\}$ in a way to know if a state $q \in Q$ of the k -coBüchi automaton is reached or not (value -1), and to know the maximal number of visits to an accepting state of runs that end up in q [13]. These counting functions are implemented as bit arrays, together with specific efficient operations implemented in C. Indeed, our algorithms manipulate antichains of counting functions, and operations like membership or intersection are not standard and cannot be implemented using existing libraries on bit arrays. (2) The simplicity of Python increases scalability and modularity and it reduces the risk of errors. Unfortunately, using Python also presents some drawbacks. Indeed, interfacing Python and C leads to light performance overhead. Nevertheless, we believe that the overhead is a small price to pay in comparison with the gain of simplicity.

Our implementation does not use BDDs, as they do not seem to be well adapted in this context, and might be outperformed by the use of antichains [17][18]. We have instead developed a library with a generic implementation of antichains that can easily be reused in another context.

Tool Download and User Interface. **Acacia+** can be downloaded at <http://lit2.ulb.ac.be/acaciaplus>. It can be installed under a single command-line version

working both on Linux and MacOsX, or used directly via a web interface. The source is licensed under the GNU General Public License. The code is open and can be used, extended and adapted by the research community. For convenience, a number of examples and benchmarks have been pre-loaded in the web interface.

Execution Parameters. Acacia+ offers many execution parameters, fully detailed in the web interface helper.

Formula. Two inputs are required: an LTL formula ϕ and a partition of the atomic signals into the sets I and O . The formula can be given as a single specification, or as a conjunction $\phi_1 \wedge \dots \wedge \phi_n$ of several specifications (for the compositional approach). Acacia+ accepts both the Wring and LTL2BA input formats, whatever the tool used to construct the automata.

Method. Formulas ϕ are processed in two steps: the first step constructs a universal co-Büchi automaton from ϕ , the second step checks for realizability (synthesis follows when ϕ is realizable). The automaton construction can be done either monolithically (a single automaton is associated with ϕ), or compositionally if the formula is given as a conjunction $\phi_1 \wedge \dots \wedge \phi_n$ (an automaton is then associated with every ϕ_i). The realizability check can be either monolithic, or compositional (only if ϕ is a conjunction of ϕ_i 's). When the automaton construction is compositional and the realizability step is monolithic, the latter starts with the union of all automata obtained from each ϕ_i .

The user can also choose between backward or forward algorithms for solving the underlying safety game. In the case of a compositional realizability check with forward option enabled, each intermediate subgame is solved backward and the whole game is solved forward. The way of parenthesizing ϕ is totally flexible: the user can specify his own parenthesizing or use predefined ones. This parenthesizing enforces the order in which the results of the subgames are combined, and may influence the performances.

Output. The output of the execution indicates if the input formula ϕ is realizable, and in this case proposes a winning strategy for the system. A winning strategy for the environment can also be returned when ϕ is unrealizable (only in case of a monolithic automaton construction). The output strategies are written in Verilog. When they are small (≤ 20 states), the corresponding Moore machines are also drawn in PNG using PyGraphviz. Many statistics about the execution are also output.

Options. For the automaton construction, the user can choose either LTL2BA [19] or Wring [20]. Both tools present advantages and drawbacks: LTL2BA works faster whereas Wring provides smaller automata. The user can also choose the starting player for the realizability game¹. We recall that the implemented algorithms are incremental; an upper bound can be imposed on the values of $k = 0, 1, 2, \dots$ used in the safety games $G(\phi, k)$. The user can also choose between either realizability check, or unrealizability check, or both in parallel. Finally Acacia+ includes several optimizations like surely losing states detection on the automata, limitation to critical signals, aso . . . All of them are enabled by default, but can be turned off.

¹ In the latter case, our tool uses the Wring module included in Lily.

² In the introduction, the realizability game has been described such that Player O , the system, plays first. A variant is to let Player I , the environment, play first.

4 Application Scenarios

In this section, we describe three typical scenarios of usage of **Acacia+**. More details and examples can be found on the website of **Acacia+**.

Controller Synthesis from LTL Specifications. A first classical use of **Acacia+** is to construct finite-state controllers that enforce LTL specifications. Such specifications are usually specified by a set of LTL assumptions on the environment, and a set of LTL guarantees to be fulfilled by the controller. Several benchmarks of synthesis problems are available for comparison with other tools: the test suite included in the tool **Lily** [7][8], a generalized buffer controller from the IBM RuleBase tutorial [21], and the load balancing system provided with the tool **Unbeast** [10][12]. The performances of **Acacia+** are better or similar to other tools, with the advantage of generating compact solutions. As an example, for the load balancing system with 4 clients, **Acacia+** first builds a universal coBüchi word automaton with 187 states, and then outputs a winning strategy as a Moore machine with 154 states. This is in contrast with the worst-case complexity analysis announcing a size exponential in 187, and with the winning strategy extracted by **Unbeast**, whose nuSMV representation is a file of 30MB. As mentioned in [11], extracting small strategies is a challenging problem.

Debugging of LTL Specifications. Writing a correct LTL specification is error prone [22]. **Acacia+** can help to debug unrealizable LTL specifications as follows. As explained in Sect. 2 when an LTL specification ϕ is unrealizable for Player O , then its negation $\neg\phi$ is realizable for Player I . A winning strategy of Player I for $\neg\phi$ can then be used to debug the specification ϕ . Again, **Acacia+** often offers the advantage to output readable compact (counter) strategies that help the specifier to correct his specification.

From LTL to Deterministic Büchi automata. As suggested to us by R. Ehlers, following an idea proposed in [6], LTL synthesis tools can be used to convert LTL formulas into an equivalent deterministic Büchi automaton (when possible). The idea is as follows. Let φ be an LTL formula over a set of signals Σ and σ be a new signal not in Σ . Let $I = \Sigma$ and $O = \{\sigma\}$. Then the formula $\phi = (\varphi \leftrightarrow GF\sigma)$ is realizable iff there exists a deterministic Büchi automaton equivalent to φ . Indeed if ϕ is realizable, then the Moore machine M representing a winning strategy for Player O can be transformed into a deterministic Büchi automaton equivalent to φ , by declaring accepting the states of M with output σ . Conversely, if there exists a deterministic Büchi automaton equivalent to φ , this automaton, outputting σ on accepting states, realizes ϕ .

Therefore one can use **Acacia+** to construct deterministic automata from LTL formulas (if possible). In [23], the author provides an automated method (together with a prototype) for the NP-complete problem of minimizing Büchi automata [24]. This method is based on a reduction to the SAT problem, and it is benchmarked on several automata obtained from a set of LTL formulas. We use those formulas to benchmark **Acacia+** on the LTL to deterministic Büchi automata problem. We obtain very short execution times and the size of the constructed automata is very close to that of a minimal deterministic Büchi automata. The minimum size is indeed reached for 18 among 26 formulas. This shows again that **Acacia+** is able to synthesize compact strategies. Finally, let us mention that a similar technique can be used to convert LTL formula into equivalent deterministic parity automata with a fixed number of priorities [6].

References

1. Acacia+, www.lit2.ulb.ac.be/acaciaplus/
2. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Principles of Programming Languages, POPL, pp. 179–190. ACM (1989)
3. Abadi, M., Lamport, L., Wolper, P.: Realizable and Unrealizable Specifications of Reactive Systems. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
4. Safra, S.: On the complexity of ω -automata. In: Foundations of Computer Science, FOCS, pp. 319–327. IEEE Computer Society (1988)
5. Pnueli, A., Rosner, R.: On the Synthesis of an Asynchronous Reactive Module. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)
6. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Foundations of Computer Science, FOCS, pp. 531–542. IEEE Computer Society (2005)
7. Lily, www.iaik.tugraz.at/content/research/design_verification/lily/
8. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Formal Methods in Computer-Aided Design, FMCAD, pp. 117–124. IEEE Computer Society (2006)
9. Schewe, S., Finkbeiner, B.: Bounded Synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)
10. Unbeast, www.react.cs.uni-sb.de/tools/unbeast/
11. Ehlers, R.: Symbolic bounded synthesis. Formal Methods in System Design 40, 232–262 (2012)
12. Ehlers, R.: Unbeast: Symbolic Bounded Synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
13. Filiot, E., Jin, N., Raskin, J.F.: Antichains and compositional algorithms for LTL synthesis. Journal of Formal Methods in System Design 39, 261–296 (2011)
14. Filiot, E., Jin, N., Raskin, F.: Exploiting structure in LTL synthesis. International Journal on Software Tools for Technology Transfer, 1–21 (2012)
15. Martin, D.: Borel determinacy. Annals of Mathematics 102, 363–371 (1975)
16. Acacia, www.lit2.ulb.ac.be/acacia/
17. 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)
18. Doyen, L., Raskin, J.-F.: Improved Algorithms for the Automata-Based Approach to Model-Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 451–465. Springer, Heidelberg (2007)
19. LTL2BA, www.1sv.ens-cachan.fr/~gastin/ltl2ba/
20. Wring, www.iaik.tugraz.at/content/research/design_verification/wring/
21. IBM RuleBase Tutorial, www.haifa.ibm.com/projects/verification/rb_homepage/tutorial3
22. Könighofer, R., Hofferek, G., Bloem, R.: Debugging unrealizable specifications with model-based diagnosis. In: Raz, O. (ed.) HVC 2010. LNCS, vol. 6504, pp. 29–45. Springer, Heidelberg (2010)
23. Ehlers, R.: Minimising Deterministic Büchi Automata Precisely Using SAT Solving. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 326–332. Springer, Heidelberg (2010)
24. Schewe, S.: Beyond hyper-minimisation - Minimising DBAs and DPAs is NP-complete. In: Theory and Applications of Satisfiability Testing, FSTTCS. LIPIcs, vol. 8, pp. 400–411. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)

MGSyn: Automatic Synthesis for Industrial Automation

Chih-Hong Cheng^{1,2,*}, Michael Geisinger^{2,*}, Harald Ruess²,
Christian Buckl², and Alois Knoll¹

¹ Department of Informatics, Technische Universität München
Boltzmannstr. 3, 85748 Garching bei München, Germany

² fortiss GmbH, Guerickestr. 25, 80805 München, Germany

Abstract. MGSyn is a programming toolbox based on game-theoretic notions of synthesis for generating production code in the domain of industrial automation. Instead of painstakingly engineering sequences of relatively low-level program code, the designer selects pre-defined hardware components together with behavioral interfaces from a given library, specifies a topology for the interconnection of components, and specifies the programming/synthesis problem in terms of what needs to be achieved. Given the model and a problem specification, MGSyn synthesizes executable C/C++ code for a concrete execution platform and an interactive simulator. The synthesized code is used to control distributed industry-standard PLCs in a FESTO modular production system.

1 Introduction

Realizing distributed process control systems with their stringent real-time and dependability, in particular safety and security requirements, is a challenging problem. The prevalent state-of-the-practice, as determined by current industrial standards including IEC 61131-3, IEC 61804, or IEC 61499, is based on painstakingly engineering sequences of relatively low-level control code using standardized libraries of function blocks. All too often this traditional style of programming leads to inefficiencies in developing and maintaining industrial production control systems, it has negative impact on the quality and dependability of the control code itself, and it results in inflexibility of production processes with prolonged start-up and changeover times.

We are proposing a new style of programming industrial automation plants based on describing *what* needs to be achieved instead of *how* these plants are actually being controlled. An example of such high-level instructions is “Drill and store red work pieces if they are facing up.” More precisely, based on capability models of hardware components and a description of what needs to be achieved, we set up a game between the hardware component controllers and the observable (sensor) environment, and synthesize a control algorithm based on a winning strategy for the controllers. MGSyn (Model, Game, Synthesis) is a tool for automating this high-level style of programming industrial automation plants by, first, synthesizing code for embedded control systems and, second, executing this code to control a distributed industrial programming logic control system.¹

* The first two authors contributed equally to this work.

¹ MGSyn is freely available under the GPLv3 license, including a step-by-step tutorial, at <http://www.fortiss.org/formal-methods>

Concerning related work, there is an ongoing interest in program synthesis; some recent works include [12][15][13][9][10][14]. For automation, models based on state-transition-diagrams have previously been used and synthesis is performed on a generalized Petri net models with input-output preservation [8][16][72]. In MGSyn, a developer provides a high-level specification for the desired behavior and the synthesis engine automatically creates a program (i.e., the state-transition-diagram) that fulfills the specification. In addition, MGSyn includes an automated deployment of such a high-level control program to lower-level executables.

We first illustrate the general modeling concepts of MGSyn in Section 2. In Section 3 we describe the deployment of executable code for our FESTO modular production² demonstrator depicted in Figure 1. This industrial automation plant is built from a set of rearrangeable hardware modules including processing stations, conveyer belts, and storage facilities, similar to the ones used in large-scale production facilities. Finally, in Section 4 we describe how the back-end synthesis engine as well as the mapping to a concrete execution platform is implemented.

2 Modeling Industrial Automation Processes with Games

Given a description of the plant and a high-level problem specification, the two-player game of program synthesis for industrial automation is played between *Controller* and *Environment*. The moves of *Controller* correspond to (i) processing actions of the plant and (ii) sensor triggering actions, whereas *Environment*'s moves determine the plant's sensor inputs, and hence uncertainty and non-determinism within the system. Winning conditions of the game are specified in terms of a subset of linear temporal logic (LTL). A particularly simple winning condition, e.g., is a set of states, which are regarded as *goals* for encoding what needs to be achieved. In this case *Controller* wins the (reachability) game if it succeeds in driving the plant towards these goals irrespective of the moves (sensor inputs) chosen by *Environment*. Such a game is specified in MGSyn from models of the hardware modules, the topology of these modules and the operational behavior for specifying *Controller* and *Environment* moves; see Section 3 for a representative model. From these ingredients, the synthesis engine of MGSyn creates intermediate, platform independent control code, which may be validated through platform-independent simulation. The deployment of executable code is based on a description of execution units and their interconnection.

Plant Modules. An industrial production plant is built up from hardware modules including conveyor belts, robot arms, or rotary plates. The key is to specify each of these components together with clearly defined *behavioral interfaces*, which may also be viewed as contracts; these contracts are respected by realizations of the interface specifications. Behavioral interfaces model the players' available moves. They consist of a list of preconditions (i.e., “When is the move legal?”) as well as effects of the move (i.e., “How does the move change the state of the system?”). Preconditions and effects make statements on *predicates* that in turn model the overall system state. An according set of predicates has to be added to the model when new moves are added.

² <http://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system/>

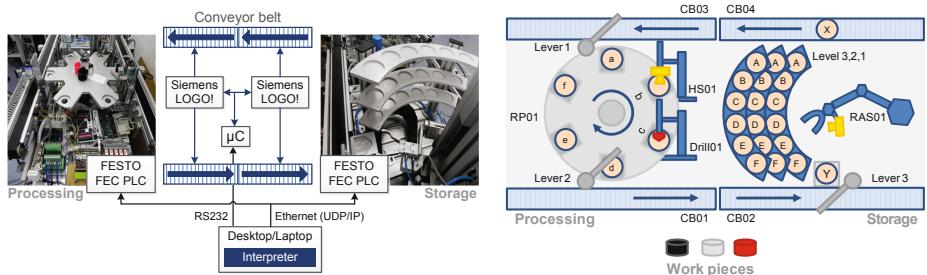


Fig. 1. The FESTO MPS demonstrator (left) and its abstract model (right)

Plant Topology. Work pieces are typically transferred between plant modules in an automation system. Therefore, we associate each hardware module with a set of *operating positions* and the topology of a plant is specified as the *overlapping of positions* among different hardware modules (see Section 3 for an example).

Plant Behavior. The so-called *problem specification* describes the desired behavior of the composed system (i.e., the goal or winning condition for *Controller*). MGSyn is restricted to specifications where controller strategies can be synthesized symbolically in time polynomial to the size of the translated game graph, ranging from reachability to *generalized reactivity-1* (GR-1) conditions [15].

Execution Platform and Networking. For generating executable program code from the intermediate, platform-independent representation as generated by the core MGSyn synthesis algorithms, it is necessary to specify the hardware setup, namely (i) the electronic control units (ECUs) that are attached to the hardware modules and (ii) the communication infrastructure.

MGSyn is implemented using the Eclipse Modeling Framework (EMF) [1] and includes an extensible library of predefined plant modules, topologies, and behavioral interfaces. Therefore, a design engineer may specify control problems by selecting modules from this library and interconnecting them in a suitable way; the corresponding game is created automatically by the MGSyn backend engine. The synthesis engine of MGSyn is based on an extension of the GAVS+ solver library [5].

3 Example: Synthesizing Executable Code for FESTO MPS

We demonstrate the use of MGSyn for modeling and synthesizing executable code for the FESTO modular production system (MPS) in Figure 1 by means of a concrete example; see also [3]. This plant consists of the modules RobotArmStorage (RAS), ConveyorBelt (CB), Lever, RotaryPlate (RP), HeightSensor (HS), and Drill.

Modeling Hardware Modules. For example, conveyor belt CB01 specifies the following properties (compare Figure 2(a)):

- Two operating positions CB01-from and CB01-to.
- Initial state belt-connected CB01-from CB01-to, where belt-connected is a predicate.

- Behavioral interface **belt-move-b** (see Figure 2(b) for formal semantics). Intuitively, the interface enables to move a work piece from position **?from** to position **?to** if the hardware supports the transmission (**belt-connected ?from ?to**). Hence, CB01 is modeled as a unidirectional belt. **at** and **occupied** are further predicates defined in the model.

We also specify ECUs and their controlled hardware modules in the model. For example, as depicted in the left part of Figure 2, FESTO FEC PLCs control the storage and processing units, while the conveyor belts are controlled by Siemens LOGO! PLCs.

Topology Specification via Overlapping Positions. Once modules are specified, the second step is to specify their topology. For instance, for the system in Figure 1, the destination of CB04 is linked with the source of CB03. This spatial overlapping is characterized in the model (see Figure 2(d)). Similarly, the source of Lever1 overlaps

(a) Project structure screenshot showing the file `T01_FESTOGarching2color.xml`. It displays the project tree with nodes like `platform/resource/MGSynTestcase/src/model/T01_FESTOGarching2color.xml`, `System Model`, `Hardware Model`, `Robot Arm Storage RAS01`, `Conveyor Belt CB01`, and `Position CB01-from`. A context menu is open over the `Conveyor Belt CB01` node, with the option `Type beltposition` highlighted.

(b) Formal semantics of the `belt-move-b` action. The code is as follows:

```

(:action belt-move-b
  :parameters (?obj - unit ?from ?to - position)
  :precondition (and (POTRAN) (belt-connected ?from ?to))
    (at ?obj ?from) (not (occupied ?to)))
  :effect (and (POTRAN) (not (occupied ?from))
    (not (at ?obj ?from))(at ?obj ?to)(occupied ?to))
)

```

(c) Sample specification (goal) in PDDL-like. The goal is to move a workpiece from its current position to a target position, ensuring it is not occupied at the target position.

```

(and
  (or (face wp1 up) (face wp1 down)) ; Height of work piece is known
  (and
    (or ; a → b ≡ ~a ∨ b
      (not face wp1 up)) ; If wp1 is facing up, then...
    (and
      (drilled wp1) ; ...ensure it is drilled and sort by color:
      (and
        (or (not (color wp1 red)) (at wp1 L1_a)) ; Red → L1_a
        (or (not (color wp1 white)) (at wp1 L2_a)) ; White → L2_a
        (or (not (color wp1 black)) (at wp1 L3_a)) ; Black → L3_a
      )
      (or ; Color of work piece is known
        (color wp1 red) (color wp1 white) (color wp1 black)
      )
    )
  )
  (or ; a → b ≡ ~a ∨ b
    (not (face wp1 down)) ; If wp1 is facing down, then...
    (and (not (drilled wp1)) (at wp1 CB03-from)) ; ...do not drill
  )
)

```

(d) Unification of two belt positions (CB04-to and CB03-from). The screenshot shows the `Topology Model` section with entries for `Position Overlapping beltposition` and `Position Overlapping beltposition`.

Property	Value
Position Unified From	Position CB04-to
Position Unified To	Position CB03-from
Unified Type	beltposition

(e) MGSyn synthesis environment. The screenshot shows the `MGSyn: Design-Synthesis-Codegen Environment` dialog box with various synthesis steps listed:

- General (Transformation) tab: `Output`
- Analysis: `Generate synthesis model (PDDL-like)` (selected), `Generate synthesis model (PDDL-like)`
- Synthesis: `Synthesize strategy in high-level control format`, `Generate architecture script from model`, `Generate executable code`

Fig. 2. (a) Unidirectional conveyor belt modeled in MGSyn. (b) Behavioral interface of **belt-move-b**. (c) Sample specification (goal). (d) Unification of two belt positions (CB04-to and CB03-from). (e) Execution of MGSyn from within Eclipse with the list of synthesis and code generation steps.

with CB03-to while its destination overlaps with RP-a, implying Lever1 is able to push work pieces from the belt to the rotary plate.

Synthesis, Architecture Mapping and Code Generation. The third step is to describe the winning condition and perform synthesis. For instance, assume that an operator (or a robot) is located at position CB03-from and feeds work pieces to the system. Due to space limits, Figure 2(c) shows only a simple specification, which is to drill and store a work piece based on its color (words after ; ; are comments). It also contains basic error handling: if the object is not facing up, it shall not be drilled and shall be returned back to the operator (position CB03-from).

To perform synthesis and code generation, the designer simply right-clicks on the EMF model to invoke MGSyn from the popup menu (Figure 2(e)). Our engine first creates a unified synthesis model by renaming overlapping positions to unique identifiers. Then the model is fed into the synthesis engine and the engine reports a winning strategy whenever possible. Synthesis for the specification in Figure 2(c) only takes seconds.

MGSyn uses a template-based approach to generate code (e.g., state variables, predicate functions, actions) from the model which is described in detail in Section 4.

4 Back-end Engine and Execution Platform Mapping

(Engine) The back-end engine first translates the EMF model to an intermediate format, which is based on PDDL [1] extended with game semantics [4]. As PDDL is very appropriate to specify behavioral interfaces, such a translation is very intuitive. However, to create a single model from multiple components, the engine needs to perform an automatic renaming over operating positions that physically overlap. Then the engine performs synthesis based on the intermediate model under the user-provided specification of the winning condition (also in a PDDL-like format). The output of the engine is a sequential program with of *Controller*'s moves, where each move is executed only if a set of conditions on the system state is true. The conditions encode the dynamic adaptation of *Controller*'s strategy to win the game in reaction to *Environment*'s moves.

The synthesis engine handles a subset of LTL properties such as GR-1 (known to be able to capture practical specifications in reactive synthesis [5]), where the complexity of game solving is polynomial in the size of the game arena, which itself is exponential in the number of Boolean variables used in system modeling. Useful optimizations for speeding up synthesis rely on analyzing the specification and identifying relevant parts of the game arena, thereby significantly reducing the number of Boolean variables in game encoding [4]. Overall, the synthesis engine of MGSyn incorporates well-known techniques for optimizing programs and bring them to assist optimizations in synthesis, such as constant propagation and cone-of-influence computations, as described in [4]. Experiments in GAVS+ confirm that the optimization techniques described in [4] often yield performance increase of at least an order of magnitude. These optimizations may also be useful as preprocessing steps for other reactive synthesis frameworks.

(Execution Platform Mapping) After the synthesis engine has generated a winning strategy for *Controller*, the strategy has to be mapped to an executable representation. This is done in two steps: First, an API matching the modeled plant modules, topology

and behavior is generated from a code template written in Xpand language using EMF tooling. Second, the synthesized solution is transformed so that it calls the functions from the API. This second step is a simple text replacement that ensures that naming conventions of the C++ programming language are enforced. The code finally compiles to a console application, which covers both simulation and execution on real hardware. For the latter scenario, the API code is based on a thin manually implemented device driver layer for triggering the respective actions on the hardware.

The presented two-step approach has the advantage that the API is independent from the actually executed strategy; it may be generated once and re-used for different winning conditions as long as the plant model remains the same.

The model elements are mapped to API code as follows: Entities (e.g., *operating positions*, work pieces, colors) are mapped to enumerations and made available as data types. *Predicates* are transformed into state variables and predicate functions for retrieving and modifying the system state. For each of *Controller's moves*, a function with the following behavior is generated: If execution on real hardware should be performed, it calls the device driver library functions to trigger the respective control action(s). It then updates the state variables according to the action's effects specified in the model. For each of *Environment's moves*, a function is generated as follows: If execution on real hardware should be performed, it calls the device driver library functions to retrieve the respective sensor reading(s). If simulation should be performed, it prints a list of possible sensor readings extracted from the model and asks the operator to make a choice. Finally, it updates the state variables according to the (simulated) sensor reading(s).

Lastly, the main program is generated as follows: When the program starts, the device drivers are initialized if execution on real hardware should be performed. Moreover, the state variables are set to their initial values according to the respective specification in the model. Then, a function representing the synthesized strategy is called, which contains the synthesis result to which the following transformations have been applied: Evaluations of predicates within the conditions of an action call predicate functions or directly evaluate state variables within the API. Invocations of control actions call the functions representing the respective moves within the API.

5 Concluding Remarks

The strengths of MGSyn show when the automation task (winning condition) needs to be adapted, as only small changes in the formal model are required to generate correct-by-construction code compared to hours or days of manual modification. In practice it is also useful that MGSyn indicates infeasibility, that is, there is no winning strategy for the specified control problem.

These features set MGSyn apart from traditional programming paradigms in automation by increasing efficiency and reducing potential sources of error. So far we have adapted MGSyn to two different FESTO MPS plants with different modules, processors, and communication protocols, which were designed according to industrial standards.

Several extensions to MGSyn are planned, including the handling of real-time properties and for incorporating basic fault-tolerance mechanisms. The injection of faults, in particular, may be modeled by moves of the *Environment*, and fault-tolerance patterns

may be incorporated into the synthesis engine as suggested in [6]. Currently, we are in the process of extending the FESTO MPS demonstrator with capabilities for communicating with the work pieces to be manufactured, which may then (compare “Internet of things / Industrie 4.0”), determine how they should be processed by the processing plant. In this way, work pieces become important new players in the game of industrial production control.

Acknowledgement. We thank Barbara Jobstmann for evaluating some optimization techniques in Anzu.

References

1. Eclipse Modeling Framework, <http://www.eclipse.org/modeling/emf/>
2. Carmona, J., Cortadella, J., Kishinevsky, M.: Genet: A tool for the synthesis and mining of petri nets. In: ACSD 2009, pp. 181–185. IEEE (2009)
3. Cheng, C.-H., Geisinger, M., Ruess, H., Buckl, C., Knoll, A.: Game solving for industrial automation and control. In: ICRA (to appear, May 2012)
4. Cheng, C.-H., Jobstmann, B., Geisinger, M., Diot-Girald, S., Knoll, A., Buckl, C., Ruess, H.: Optimizations for game-based synthesis. Technical Report 12, Verimag (2011)
5. Cheng, C.-H., Knoll, A., Luttenberger, M., Buckl, C.: GAVS+: An Open Platform for the Research of Algorithmic Game Solving. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 258–261. Springer, Heidelberg (2011)
6. Cheng, C.-H., Rueß, H., Knoll, A., Buckl, C.: Synthesis of Fault-Tolerant Embedded Systems Using Games: From Theory to Practice. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 118–133. Springer, Heidelberg (2011)
7. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Transactions on Information and Systems 80(315-325), 182 (1997)
8. Der Jeng, M., DiCesare, F.: A review of synthesis techniques for petri nets with applications to automated manufacturing systems. IEEE Transactions on Systems, Man and Cybernetics 23(1), 301–312 (1993)
9. Dimitrova, R., Finkbeiner, B.: Synthesis of Fault-Tolerant Distributed Systems. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 321–336. Springer, Heidelberg (2009)
10. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
11. Ghallab, M., Aeronautiques, C., Isi, C., Penberthy, S., Smith, D., Sun, Y., Weld, D.: PDDL—the planning domain definition language. Technical Report CVC TR-98003/DCS TR-1165, Yale Center for Computer Vision and Control (October 1998)
12. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD 2006, pp. 117–124. IEEE (2006)
13. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A Tool for Property Synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
14. Madhusudan, P.: Synthesizing reactive programs. In: CSL 2011. LIPIcs, vol. 12, pp. 428–442. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
15. 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)
16. Uzam, M., Zhou, M.: An iterative synthesis approach to petri net-based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 37(3), 362–371 (2007)

OpenNWA: A Nested-Word Automaton Library*

Evan Driscoll¹, Aditya Thakur¹, and Thomas Reps^{1,2}

¹ Computer Sciences Department, University of Wisconsin – Madison

{driscoll,adi,reps}@cs.wisc.edu

² GrammaTech, Inc., Ithaca, NY

Abstract. Nested-word automata (NWAs) are a language formalism that helps bridge the gap between finite-state automata and pushdown automata. NWAs can express some context-free properties, such as parenthesis matching, yet retain all the desirable closure characteristics of finite-state automata.

This paper describes OpenNWA, a C++ library for working with NWAs. The library provides the expected automata-theoretic operations, such as intersection, determinization, and complementation. It is packaged with WALi—the Weighted Automaton Library—and interoperates closely with the weighted pushdown system portions of WALi.

1 Introduction

Many problems in computer science are solved by modeling a component as an automaton. Traditionally, this means either confronting several undecidable problems that arise with the use of pushdown automata or giving up expressivity, and usually precision, by using finite-state automata. Recently, the development of nested word automata (NWAs) and related formalisms [23] has revealed a fertile middle ground between these two extremes. NWAs are powerful enough to express some “context-free”-style properties, such as parenthesis matching, and yet retain the decidability properties that make it convenient to work with regular languages. In particular, NWAs and their languages are closed under operations such as intersection and complementation. NWAs have been applied in areas such as modeling programs and XML documents. When modeling programs, NWAs can eliminate spurious data flows along paths with mismatched calls and returns. In XML documents, NWAs can model the matching between opening and closing tags.

We have created a C++ implementation of NWAs, called OpenNWA. OpenNWA is packaged with the *Weighted Automata Library*, WALi [7]. WALi also provides implementations for weighted finite-state automata and weighted pushdown systems (WPDSSs). The OpenNWA library

* Supported by NSF under grants CCF-{0540955, 0810053, 0904371}; by ONR under grants N00014-{09-1-0510, 10-M-0251}; by ARL under grant W911NF-09-1-0413; and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsoring agencies.

- implements (in the terminology of [3]) linearly-accepting, weakly-hierarchical NWAs, along with standard automata-theoretic operations. (See §3.1)
- is extensible via a mechanism that allows the user to attach arbitrary client information to each node in the automaton. (See §3.2)
- inter-operates with WALI’s WPDS library and allows the user to issue queries about an NWA’s configuration space. (See §3.3)
- provides utilities for operating on a textual NWA format [5, §5].
- provides extensive documentation [5] and a test suite.

OpenNWA is currently used by three projects [11, 4, 6]; see §4. OpenNWA is available at <http://research.cs.wisc.edu/wpis/OpenNWA>.

2 NWAs

This section describes nested-word automata [2] and related terms at an intuitive level, and gives an example of how they are used in program analysis. For the formal definitions that we use, see our technical report [5, App. A].

A nested word is an ordinary (linear) string of symbols over some alphabet Σ paired with a *nesting relation*. The nesting relation describes a hierarchical relation between input positions, for instance between matched parentheses. Graphically, a nested word can be depicted as illustrated to the right. In this image, following just the horizontal arrows illustrates the linear word, while the curved edges (“nesting edges”) indicate positions that are related by the nesting relation. For a nesting relation to be valid, nesting edges must only point forward in the word and may not share a position or cross.

Positions in the word that appear at the left end of a nesting edge are called *call positions*, those that appear at the right end are called *return positions*, and the remaining are *internal positions*. It is possible to have *pending* calls and returns, which are not matched within the word itself. For a given return, the source of the incoming nesting edge is called that return’s *call predecessor*.

Nested-word automata (NWAs) are a generalization of ordinary finite-state automata (FA). An NWA’s transitions are split into three sets—call transitions, internal transitions, and return transitions. Call and internal transitions are defined the same as transitions in ordinary FAs, while return transitions also have a call-predecessor state as an additional element.

To understand how an NWA works, consider first the case of an ordinary FA M . We can think of M ’s operation as labeling each edge in the input word with the state that M is in after reading the symbol at that edge’s source. For instance, shown to the right is a partial run. To find the next state, M looks for a transition out of q_2 with the symbol c —say with a target of q_3 —and labels the next edge with q_3 .

The operation of an NWA proceeds in a fashion similar to a standard FA, except that the machine also labels the nesting edges. When the NWA reads an internal position, it chooses a transition and labels the next linear edge the same way an FA would. When the NWA reads a call position, it picks a matching call

```

1 void main() {
2     f = factorial(5);
3     printf("%d\n", f);
4 }

5 int factorial(int n) {
6     if (n == 0)
7         return 1;
8     f = factorial(n-1);
9     return n * f;
10}

```

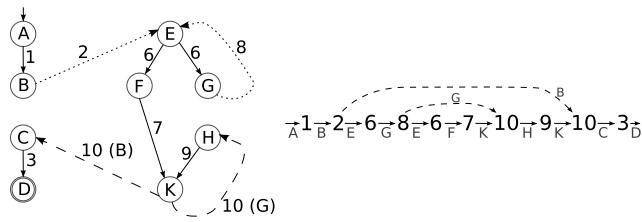


Fig. 1. An example program, corresponding NWA, and accepted word. State labels are arbitrary; transition symbols give the line number of the corresponding statement. Some nodes are elided. Dotted lines indicate call transitions, and dashed lines are return transitions. The state in parentheses on a return transition is the call predecessor.

transition and labels the next linear edge in the same way, but also labels the outgoing *nesting* edge with the state that the NWA is leaving. When the NWA reads a return position, it looks not only at the preceeding linear state but also at the state on the incoming nesting edge. It then chooses a return transition that matches both, and labels the next linear edge with the target state. An example NWA run is shown in Fig. II

OpenNWA supports ε internal transitions, which operate in an analogous way to ε rules in ordinary FAs. It also supports something we call *wild* transitions, which match any single symbol. Wilds can appear on any transition type.

Example 1. NWAs can be used to encode the interprocedural control-flow graph (ICFG) of a program. Intraprocedural ICFG edges become internal transitions, interprocedural call edges become call transitions, and interprocedural return edges become return transitions. For an ICFG return edge (*exit-site*, *return-site*), we use the call site that corresponds to *return-site* in the call-predecessor position of the NWA’s transition. The symbols on a transition depend on the application, but frequently are the corresponding statement.

An example program, the corresponding NWA, and an example word accepted by that NWA are shown in Fig. II. Using an NWA allows us to exclude paths such as 1-2-5-6-7-8-9-... that do not follow a matched path of calls and returns. Fewer paths can allow a client analysis to obtain increased precision.

3 The OpenNWA library

OpenNWA provides a C++ class called `NWA` for representing NWAs. For information about constructing NWAs and actual API information, see the OpenNWA documentation [5]. In this section, we briefly describe some things a user can do with an NWA (or NWAs) once it is built.

3.1 Automata-Theoretic Operations

As mentioned in §II, OpenNWA supports most automata-theoretic operations:

- intersection
- reversal
- complement
- union
- concatenation
- emptiness checking
- Kleene star
- determination
- example word generation

For the most part, we use Alur and Madhusudan’s algorithms [23]; however we note three exceptions. First, we implemented emptiness checking and example generation using WPDSs, discussed below. Second, we found and corrected a minor error in Kleene star [5, §6.4]. Third, we expressed Alur and Madhusudan’s determinization algorithm using relational operators [5, App. B].

OpenNWA supports determining whether an NWA’s language is empty, and if it is not, OpenNWA can return an arbitrary word from the NWA’s language. It is also possible to ask specifically for a shortest accepted word. The library performs these operations by converting the NWA into a WPDS and running a post^* query from the set of initial configurations (see §3.3). To find an example word, OpenNWA uses a witness-tracing version of post^* [10, §3.2.1], and extracts the word from the resulting witness. For the shortest word, we simply use weights that track the length of the path from the initial state.

The ability to obtain an example word is important in program analysis. It can show the end user of an analysis tool a program trace that may violate a property. Moreover, this feature is fundamental to counterexample-guided refinement: in CEGAR-based model checkers, the counterexample is typically an example word from the automaton that models the program.

3.2 Client Information

OpenNWA provides a facility that we call *client information*. This feature allows the user of the library to attach arbitrary information to each state in the NWA. For instance, as discussed in §4, McVeto uses NWAs internally, and uses client information to attach a formula to each state in the NWA.

The library tracks this information as best as it can through each of the operations discussed in the previous section, and supports callback functions to compute new client information when it does not have the information it needs.

3.3 Inter-operability with WPDSs

Weighted pushdown systems (WPDSs) can be used to perform interprocedural static analysis of programs [10]. The PDS proper provides a model of the program’s control flow, while the weights on PDS rules express the dataflow transformers. Algorithms exist to query the configuration space of WPDSs, which corresponds to asking a question about the data values that can arise at a set of configurations in the program’s state space. A configuration consists of a control location and a list of items on the stack.

OpenNWA supports converting an NWA into a WALi WPDS. This feature allows an OpenNWA client to issue queries about the configuration space of an NWA. The WPDS’s stack corresponds to the states that label the as-yet-unmatched nesting edges in a prefix of an input nested word. When viewed in program-analysis terms, the WPDS that results from this conversion reflects the same control structure as the original NWA. Answers to WPDS queries tend to have a natural interpretation in the NWA world, as well. For NWA operations that can be expressed naturally in this way (e.g., emptiness checking and post^*), OpenNWA employs this conversion. Other NWA operations, such as determinization, do not have an equivalent WPDS operation.

NWAs themselves are not weighted, but OpenNWA provides a facility for determining the weights of the WPDS rules during conversion. The user provides an instance of class `WeightGen`, which acts as a factory function. The function is called with the states in question and returns the weight of the resulting WPDS rule. (The weight can depend on the client information of the states.)

4 Uses of OpenNWA

I/O Compatibility Checking. We used OpenNWA as the primary component of a tool called the Producer-Consumer Conformance Analyzer (PCCA) [4]. Given two programs that operate in a producer/consumer relationship over a stream, PCCA’s goal is to determine whether the consumer is prepared to accept all messages that the producer can emit, and find a counterexample if not. PCCA infers a model of the output language of the producer, infers a model of the input language of the consumer, and determines whether the models are compatible.

PCCA uses NWAs for its models, building them from the ICFG, as discussed in Ex. II. Edges corresponding to statements that perform I/O are labeled with the type of the I/O, and all other internal transitions are labeled with ε . Conceptually what we want to check is whether the producer’s output language is a subset of the consumer’s input language, which is an operation NWAs and our library support. In practice, this check is likely to be too strict, and we need an additional step, detailed in [4, §2.3 and §3.2].

Machine-Code Model Checking. McVeto is a machine-code verification engine [II] that, given a binary and description of a bad target state, tries to find either (i) an input that forces the bad state to be reached, or (ii) a proof that the bad state is impossible to reach.

McVeto uses a model of the program called a *proof graph*, which is an NWA that overapproximates the program’s behavior. States in a proof graph are labeled with formulas; edges are labeled with program statements. McVeto starts with a very coarse overapproximation, which it then refines. One refinement technique uses symbolic execution to produce a concrete trace of the program’s behavior, performs *trace generalization* [II, §3.1] to convert the trace into an overapproximating NWA (the “folded trace”), and intersects the current proof graph with the folded trace to obtain the new proof graph. The formula on a state in the new proof graph is the conjunction of the formulas on the states that are being paired from the current proof graph and the folded trace.

McVeto’s implementation uses OpenNWA’s client-information feature to store the formula for each state. During intersection, the callback functions mentioned in §3.2 compute the conjunction of the input formulas, which are then used in the new proof graph.

To determine whether the target state is not reachable in the proof graph (and thus is definitely not reachable in the actual program), McVeto calls `prestar()` (see §3.3). The result of this call is also used to determine which “frontier” to extend next during directed test generation [II, §3.3].

JavaScript Security-Policy Checking and Weaving. The JAM tool [6] checks a JavaScript program against a security policy, either verifying that the program is already correct with respect to that policy or inserting dynamic checks into the program to ensure that it will behave correctly. JAM builds *two* models of the input program, one that overapproximates the control flow of the program and one that overapproximates the data flow. The policy is also expressed as an NWA of forbidden traces. By intersecting the policy automaton with both program models, JAM obtains an NWA that expresses traces that possibly violate the policy.

Once JAM has the combined NWA, it asks OpenNWA for a shortest word in the language. If the language is empty (i.e., there is no shortest word), the program always respects the policy. If OpenNWA returns an example word w , JAM checks whether w corresponds to a valid trace through the program. If w is valid, then JAM inserts a dynamic check to halt concrete executions corresponding to w that would violate the policy. If w is not valid, than JAM can either refine the abstraction and repeat, or insert a dynamic check to detect and halt concrete executions corresponding to w for which the policy would be violated.

5 Related work

Alur and Madhusudan each maintain a page giving a significant bibliography of papers that present theoretical results, practical applications, and tools related to NWAs and visibly pushdown automata (VPAs) [18]. VPAs and their languages are another formalism which can be seen as an alternative encoding of NWAs and nested-word languages [3].

VPAlib [9] is a general-purpose library implementing VPAs. However, OpenNWA’s implementation is far more complete. For instance, VPAlib does not support concatenation, complementation (although it does support determinization), checking emptiness, or obtaining an example word.

References

1. Alur, R.: Nested words (2011), <http://www.cis.upenn.edu/~alur/nw.html>
2. Alur, R., Madhusudan, P.: Adding Nesting Structure to Words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 1–13. Springer, Heidelberg (2006)
3. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56(3) (May 2009)
4. Driscoll, E., Burton, A., Reps, T.: Checking conformance of a producer and a consumer. In: FSE (2011)
5. Driscoll, E., Thakur, A., Burton, A., Reps, T.: WALi: Nested-word automata. TR-1675R, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (September 2011)
6. Fredrikson, M., Joiner, R., Jha, S., Reps, T., Porras, P., Saïdi, H., Yegneswaran, V.: Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 548–563. Springer, Heidelberg (2012)
7. Kidd, N., Lal, A., Reps, T.: WALi: The Weighted Automaton Library (2007), <http://www.cs.wisc.edu/wpis/wpds/download.php>

8. Madhusudan, P.: Visibly pushdown automata – automata on nested words (2009),
<http://www.cs.uiuc.edu/~madhu/vpa/>
9. Nguyen, H.: Visibly pushdown automata library (2006),
<http://www.emn.fr/z-info/hnguyen/vpa/>
10. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP 58(1-2) (October 2005)
11. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. TR 1669, UW-Madison (April 2010); Abridged version published in CAV 2010

UFO: A Framework for Abstraction- and Interpolation-Based Software Verification

Aws Albarghouthi¹, Yi Li¹, Arie Gurfinkel², and Marsha Chechik¹

¹ Department of Computer Science, University of Toronto, Canada

² Software Engineering Institute, Carnegie Mellon University, USA

Abstract. In this paper, we present UFO, a framework and a tool for verifying (and finding bugs in) sequential C programs. The framework is built on top of the LLVM compiler infrastructure and is targeted at researchers designing and experimenting with verification algorithms. It allows definition of different abstract post operators, refinement strategies and exploration strategies. We have built three instantiations of the framework: a predicate abstraction-based version, an interpolation-based version, and a combined version which uses a novel and powerful combination of interpolation-based and predicate abstraction-based algorithms.

1 Introduction

Software model checking tools prove that programs satisfy a given safety property by computing inductive invariants that preclude erroneous program states. Over the past decade, software model checking tools have adopted a number of different techniques for computing invariants which we categorize as *Overapproximation-Driven* (OD) and *Under-approximation-driven* (UD).

OD tools, e.g., SLAM [4], BLAST [6], and SATABS [10], utilize an abstract domain based on predicate abstraction [11] to compute an over-approximation of the set of reachable states of a program. In the case of false positives, such techniques employ an abstraction refinement loop [9] to refine the abstract domain and eliminate false positives.

UD tools, spearheaded by IMPACT [15] and YOGI [17], compute invariants by generalizing from infeasible symbolic program paths, thus bypassing the potentially expensive computation of the abstract post operator. For example, IMPACT and WOLVERINE [13] use Craig interpolants, extracted from the proofs of unsatisfiability of formulas encoding an infeasible path to error, in order to eliminate a potentially large number of paths to error and prove a program safe. WHALE [2] extends IMPACT to the interprocedural case by using under-approximations of functions to compute function summaries. Similarly, YOGI uses weakest-preconditions (instead of interpolants) along infeasible program paths, chosen based on concrete test-case executions, in order to strengthen a partition-graph of the state space of a program.

In this paper, we present UFO, a framework and a tool for *verifying* and *falsifying* safety properties of sequential C programs. The features of UFO are:

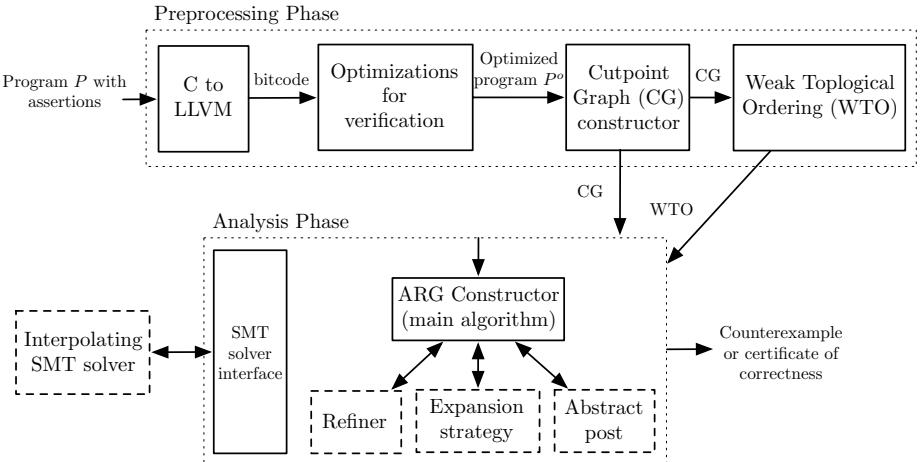


Fig. 1. The architecture of UFO

1. It is a framework for building and experimenting with UD, OD, and combined UD/OD verification algorithms. It is *parameterized* by the abstract post operator, refinement strategy, and expansion strategy.
2. It comes with a number of instantiations whose novel features are described in [1]: several predicate abstraction-based OD instantiations, an interpolation-based UD instantiation, and several combined OD/UD instantiations that use different forms of predicate abstraction to augment and strengthen interpolation-based analysis, a technique that we have shown to be quite powerful in [1]. To the best of our knowledge, UFO is the first available tool to implement a combined UD/OD algorithm. Moreover, these instantiation of UFO implement a novel interpolation-based refinement strategy that computes interpolants (abstractions) for multiple program paths encoded in a single SMT formula. That is, unlike other tools that enumerate paths explicitly, e.g., [15][13][6], UFO delegates path enumeration to an SMT solver.
3. It is implemented on top of the open-source LLVM compiler infrastructure [14]. Since LLVM is a well-maintained, well-documented, and continuously improving framework, it allows UFO users to easily integrate program analyses, transformations, and other tools built on LLVM (e.g., KLEE [8]), as they become available. Furthermore, since UFO analyzes LLVM bitcode (intermediate language), it is possible to experiment with verifying programs written in other languages compilable to LLVM bitcode, such as C++, Ada, and Fortran.

The architecture and parameterization of UFO and the underlying LLVM framework provide users with an extensible environment for experimenting with different software verification algorithms.

UFO is available at <http://www.cs.toronto.edu/~aws/ufo>.

2 The Implementation and Architecture of UFO

UFO is implemented on top of the LLVM compiler infrastructure [14] – see Figure 1 for an architectural overview. UFO accepts as input a C program P with assertions. For simplicity of presentation, let $P = (V, T, \phi_I, \phi_E)$, where V is the set of program variables, T is the transition relation of the program (over V and V' , the set of primed variables), ϕ_I is a formula describing the set of initial states, and ϕ_E is a formula describing the set of error states.

First, P goes through a *preprocessing phase* where it is compiled into LLVM bitcode (intermediate representation) and optimized for verification purposes, resulting in a semantically equivalent but optimized program $P^o = (V^o, T^o, \phi_I^o, \phi_E^o)$.

Then, the *analysis phase* verifies P^o and either outputs a certificate of correctness or a counterexample. A certificate of correctness for P^o is a safe inductive invariant I s.t. (1) $\phi_I^o \Rightarrow I$, (2) $I \wedge T^o \Rightarrow I'$, and (3) $I \wedge \phi_E^o$ is UNSAT.

2.1 Preprocessing Phase

We now describe the various components of the preprocessing phase.

C to LLVM. The first step converts the program P to LLVM bitcode using the `llvm-gcc` or `clang` compilers.

Optimizations for Verification. A number of native LLVM optimizations are then applied to the bitcode, the most important of which are *function inlining (inline)* and *static single assignment (SSA) conversion (mem2reg)*. Since UFO implements an intraprocedural analysis, it requires all functions to be inlined into `main`. In order to exploit efficient SMT program encoding techniques like [12],

UFO expects the program to be in SSA form. A number of standard program simplifications are also performed at this stage, with the goal of simplifying verification. The final result is the optimized program P^o . Mapping counterexamples from P^o back to the original C program P is made possible by the debugging information inserted into the generated bitcode by `clang`.

Before the above optimizations could be applied, we had to bridge the gap between the semantics of C assumed by LLVM (built for compiler construction) and the verification benchmarks. Consider, for example, the program in Figure 2. After LLVM optimizations, it is reduced to the empty program: `return 1;`. LLVM replaces undefined values by constants that result in the simplest possible program. In our example, the conditions of both `if`-statements are assigned to 0, even though they contradict each other. On the other hand, verification benchmarks such as [5] assume that without an explicit initialization, the value of `x` is non-deterministic. To account for such semantic differences, a UFO-specific LLVM transformation is scheduled before optimizations are run. It initializes each variable with a call to an external function `nondet()`, forcing LLVM not to make assumptions about its value.

```
int x;
if (x == 0)
    func1();
if (x != 0)
    func2();
return 1;
```

Fig. 2. Example program.

Cutpoint Graph and Weak Topological Ordering. A *cutpoint graph* (CG) is a “summarized” control-flow graph (CFG), where each node represents a cutpoint (loop head) and each edge represents a loop-free path through the CFG. Computed using the technique presented in [12], the CG is used as the main representation of the program P^o . Using it allows us to perform abstract post operations on loop-free segments, utilizing the SMT solver (e.g., in the case of predicate abstraction) for enumerating a potentially exponential number of paths. A *weak topological ordering* (WTO) [7] is an ordering of the nodes of the CG that enables exploring it with a *recursive iteration strategy*: starting with the inner-most loops and ending with the outer-most ones.

2.2 Analysis Phase

The analysis phase, which receives the CG and the WTO of P^o from the pre-processing phase, is comprised of the following components:

ARG Constructor. The *ARG Constructor* is the main driver of the analysis. It maintains an *abstract reachability graph* (ARG) [1] of the CG – an unrolling of the CG, annotated with formulas representing over-approximations of reachable states at each cutpoint. ARGs can be seen as DAG representations of *abstract reachability trees* (ARTs) used in lazy abstraction [15,6]. When the algorithm terminates without finding a counterexample, the annotated ARG represents a certificate of correctness in the form of a safe inductive invariant I for P^o . To compute annotations for the ARG, the ARG constructor uses three *parameterized components*: (1) the abstract post, to annotate the ARG as it is being expanded; (2) the refiner, to compute annotations that eliminate spurious counterexamples; and (3) the expansion strategy, to decide where to restart expanding the ARG after refinement.

Abstract Post. The abstract post component takes a CG edge and a formula ϕ_{pre} describing a set of states, and returns a formula ϕ_{post} over-approximating the states reachable from ϕ_{pre} after executing the CG edge. UFO includes two common implementations of abstract post – Boolean and Cartesian predicate abstractions [3].

Refiner. The refiner receives the current ARG with potential paths to an error location (i.e., the error location is not annotated with *false*). Its goal is either to find a new annotation for the ARG s.t. the error location is annotated with *false*, or to report a counterexample. UFO includes an interpolation-based implementation of the refiner.

Expansion Strategy. After the refinement, the ARG constructor needs to decide where to restart expanding the ARG. The *expansion strategy* specifies this parameter. UFO includes an eager strategy and a lazy strategy, both of which are described in the following section.

SMT Solver Interface. Components of the analysis phase use an SMT solver in a variety of ways: (1) The ARG constructor uses it to check that the annotations of the ARG form a safe inductive invariant; (2) abstract post, e.g., using predicate abstraction, encode post computations as SMT queries, and (3) the refiner can use it to find counterexamples and to compute interpolants. All these uses are

handled through a general interface to two SMT solvers: MATHSAT5¹ (used for interpolation) and Z3 [16] (used for quantifier elimination).

3 Instantiations of UFO

We have implemented the three instantiations of the algorithm of [1] in UFO: (1) an interpolation-based UD instantiation, (2) a predicate abstraction-based OD instantiation, and (3) a combined OD/UD instantiation that uses predicate abstraction to augment the interpolation-based analysis. In [1], we showed that the combined instantiation can outperform both the UD and the OD instantiations. All of these instantiations use a novel interpolation-based refinement strategy where all paths in the ARG are encoded as a single SMT solver formula, delegating path enumeration to the SMT solver instead of enumerating them explicitly as done by IMPACT [15] and YOGI [17].

We now show how these instantiations are produced by defining the three UFO parameters: abstract post, refiner, and expansion strategy.

UD Instantiation. In the UD case, the abstract post always returns *true*, the weakest possible over-approximation. The annotations returned by the refiner are used as for the ARG; therefore, they can be seen as a guess of the safe inductive invariant I . If the guess does not hold, i.e., it is not inductive, then the *lazy* expansion strategy starts expanding the ARG from the inner-most loop where the guess fails [1]. The ARG is then extended and a new guess for I is made by the refiner.

OD Instantiation. In the OD case, the abstract post is based on either Boolean or Cartesian predicate abstraction. The annotations returned by the refiner are used to update the set of predicates but not to guess invariants (and thus annotate the ARG) as in the UD case. The expansion strategy used is *eager*: expansion is restarted from the root of the ARG, i.e., in each iteration UFO computes an abstract post fixpoint from the initial states ϕ_I , but with a larger set of predicates from the one used in the previous iteration.

Combined UD/OD Instantiation. In the combined UD/OD case, UFO uses Boolean or Cartesian predicate abstraction [3] to improve guesses of I found through interpolants. In each iteration, UFO starts with a guess I , that does not hold, from the previous iteration. A new set of states I_p , where $I \Rightarrow I_p$, is computed by applying an abstract fixpoint computation, based on predicate abstraction, starting from the set of states I and using the transition relation T . Technically, this is performed by expanding the ARG where the guess I fails (as in the UD case). If I_p is not a safe inductive invariant, a new guess is computed using interpolants, and the process is restarted. The trade-off in this case is between the potential for computing invariants in fewer refinements (guesses) using predicate abstraction and the potentially high cost of predicate abstraction computations.

¹ <http://mathsat.fbk.eu>

4 Conclusion

In this paper, we have described UFO, a framework and a tool for software verification of sequential C programs. As we have shown, by varying the parameters of UFO, it can be instantiated into tools employing varying verification techniques, including an interpolation-based tool, a predicate abstraction-based one, and a tool that combines the two techniques.

UFO's architecture and the fact that is built on top of LLVM provide verification algorithm designers with a flexible and extensible platform to experiment with a wide variety of verification algorithms.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-Approximations to Over-Approximations and Back. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 157–172. Springer, Heidelberg (2012)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: WHALE: An Interpolation-Based Algorithm for Inter-procedural Verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 39–55. Springer, Heidelberg (2012)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
4. 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)
5. Beyer, D.: Competition On Software Verification (2012), <http://sv-comp.sosy-lab.org/>
6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker BLAST. STTT 9(5-6), 505–525 (2007)
7. Bourdoncle, F.: Efficient Chaotic Iteration Strategies with Widening. In: Pottosin, I.V., Bjørner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
8. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Proc. of OSDI 2008, pp. 209–224 (2008)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
10. Clarke, E., Krong, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
11. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
12. Gurfinkel, A., Chaki, S., Sapra, S.: Efficient Predicate Abstraction of Program Summaries. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 131–145. Springer, Heidelberg (2011)
13. Kroening, D., Weissenbacher, G.: Interpolation-Based Software Verification with WOLVERINE. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)

14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004 (2004)
15. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
16. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The YOGI Project: Software Property Checking via Static Analysis and Testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)

SAFARI: SMT-Based Abstraction for Arrays with Interpolants

Francesco Alberti¹, Roberto Bruttomesso², Silvio Ghilardi², Silvio Ranise³,
and Natasha Sharygina¹

¹ Formal Verification and Security Lab, University of Lugano, Switzerland

² Università degli Studi di Milano, Milan, Italy

³ FBK-Irst, Trento, Italy

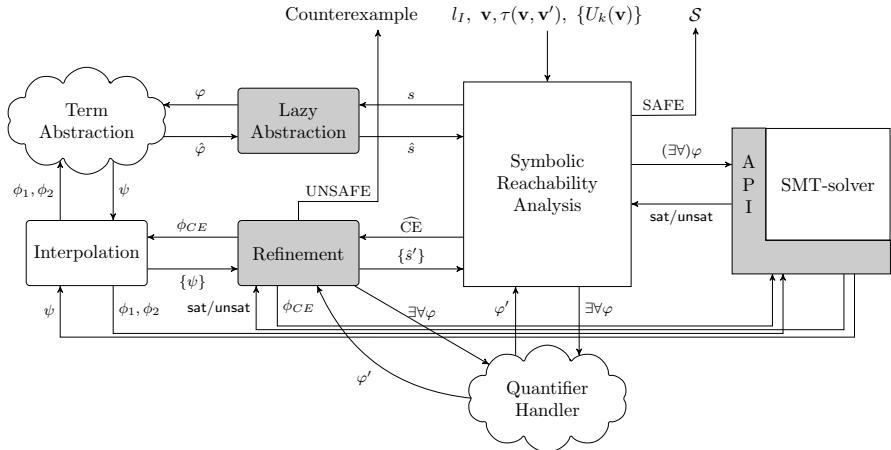
Abstract. We present SAFARI, a model checker designed to prove (possibly universally quantified) safety properties of imperative programs with arrays of unknown length. SAFARI is based on an extension of lazy abstraction capable of handling existentially quantified formulæ for symbolically representing states. A heuristics, called term abstraction, favors the convergence of the tool by “tuning” interpolants and guessing additional quantified variables of invariants to prune the search space efficiently.

1 Introduction

Efficient and automatic static analysis of imperative programs is still an open challenge. A promising line of research investigates the use of model-checking coupled with abstraction-refinement techniques [2,5,8,10,14,15] including Lazy Abstraction [3,12] and its later improvements that use interpolants during refinement [13]. An intrinsic limitation of the approaches based on Lazy Abstraction is that they manipulate quantifier-free formulæ to symbolically represent states. However, when defining properties over arrays, universal quantified formulæ are needed, e.g., as in specifying the property “the array is sorted”. The tool we present, SAFARI, is based on a novel approach [1], in which Lazy Abstraction is used in combination with the backward reachability analysis behind the Model Checking Modulo Theories (MCMT) framework [9]. The resulting procedure allows checking safety properties for arrays that require universal quantification over the indices. Moreover, the presence of quantifiers requires particular care when computing interpolants. SAFARI comes with an efficient quantifier handling procedure, exploited to retrieve quantifier-free interpolation queries from instantiations of pairs of inconsistent quantified formulæ.

The paper presents the tool architecture and the implementation details such as heuristics for abstraction, interpolation tuning, quantifier handling, and synthesis of additional quantified variables in invariants.

Many efficient tools for imperative programs verification have been developed so far. The main difference between SAFARI and other model-checkers (e.g., BLAST [3], IMPACT [13] and MAGIC [4]) is the ability of handling unbounded

**Fig. 1.** Tool architecture

arrays. Unlike ACSAR [14], SAFARI is able to discover new quantified predicates. Our approach does not require templates for predicate-discovering [16] and differs from abstract-interpretation techniques (e.g., [6, 7, 11]) in that it is based on a declarative framework that allows for identifying classes of array programs on which the core procedure terminates. For programs not meeting termination hypothesis user may suggest hints to SAFARI by means of accurate term abstraction lists as to help the tool to converge.

2 The Tool

The tool architecture is sketched in Fig. 1. SAFARI takes as input a transition system $(\mathbf{v}, \tau(\mathbf{v}, \mathbf{v}'))$ representing the encoding of an imperative program: \mathbf{v} is the set of state variables among which some are arrays, and it always contains a variable pc ranging over a finite set $\{l_0, \dots, l_n\}$ of program locations, among which we distinguish an initial location l_I .¹ A set of formulæ $\{U_k(\mathbf{v})\}$ representing unsafe states is also given to the tool; each U_k represents a violation of an assertion in the code. Next we describe the main modules of the tool.

Symbolic Reachability Analysis - This module implements a classical backward reachability analysis. Starting from the set of unsafe states, it repeatedly computes the pre-images with respect to the transition relation. It halts once it finds (the negation of a) *safe inductive invariant* \mathcal{S} for the input system or when a run from the initial state to an unsafe state is found. The symbolic reachability search is based on the *safety* and the *covering* tests: the former checks the violation of an assertion while the latter implements the fix-point condition.

Lazy Abstraction - The search for a safe inductive invariant on the original (concrete) system may require a lot of resources or it cannot be computed because of possible divergence. To mitigate this problem, SAFARI relies on the

¹ The reader is referred to [1] for details.

Lazy Abstraction paradigm: in particular it extends it by allowing existentially quantified formulae to represent states involving arrays. Moreover, SAFARI is able to introduce new quantified predicates on the fly, by means of *Term Abstraction* as described later on.

Quantifier Handling - The presence of quantified formulae imposes particular attention during the satisfiability tests: available SMT-Solvers might not be able to automatically find suitable instances for the quantified variables. SAFARI provides a specific instantiation procedure, adapted from [9] to address this issue. To be effective, this procedure implements caching of information inside of specific data-structures used to represent formulae. On one hand the caching increases the amount of space, on the other hand it cuts the number of instantiations due to constant-time checks. Alternatively, the quantified query may be passed to the SMT-Solver directly.

Refinement - This module receives an abstract counterexample and it checks first if the counterexample has a concrete counterpart. If so a feasible execution violating an assertion U_k is returned to the user. Otherwise the formulae representing the states along the abstract execution trace have to be strengthened, possibly by adding new predicates, in order to rule out spurious executions. In the current implementation, refinement is performed by means of interpolation: the *Refinement* module iteratively interacts with the *Interpolation* module in order to retrieve quantifier-free interpolants.

Interpolation - Quantifier-free interpolation for formulae involving arrays is in general not possible: in our case this situation is complicated by the presence of existential quantifiers. However, in [1], we show that the particular structure of the queries we handle, admits an equisatisfiable formulation at the quantifier-free level, for which meaningful quantifier-free interpolants can be computed. Quantifiers can be then reintroduced back, to preserve the original semantic of the formulae. This technique, however, may not be sufficient to discover suitable new quantified predicates. To address this problem, SAFARI combines interpolation with a procedure called *Term Abstraction*.

Term Abstraction - Term Abstraction is a novel technique applied during the abstraction phase to select the “right” overapproximation to be computed, and during the refinement phase to “lift” the concrete infeasible counterexample to a more abstract level, by eliminating some terms. The effect of Term Abstraction is that of controlling both the abstraction function and the interpolants produced during refinement. Term Abstraction is discussed in detail in Section 3.

SMT-Solver - The tool relies on an SMT-Solver to decide satisfiability queries. An abstract interface provides an API to separate the actual SMT-Solver used and the services which are requested by SAFARI. This interface allows the invocation of different engines needed for particular tasks. SAFARI provides interface for OPENSMT and SMT-LIB v.2.

Implementation - SAFARI is written in C++ and can be downloaded from <http://verify.inf.usi.ch/safari>. Information on the usage of the tool and a full description of all the options can be found on the SAFARI’s website.

3 Discussion

Term Abstraction and Its Benefits. Term Abstraction is the main heuristic which distinguishes SAFARI from other tools based on abstraction-refinement. It works as follows. Suppose we are given an unsatisfiable formula of the form $\psi_1 \wedge \psi_2$, and a list of undesired terms t_1, \dots, t_n (called *term abstraction list*). The underlying idea is that terms in this list should be abstracted away for achieving convergence of the model checker. Iteratively we check if $\psi_1(c_i/t_i) \wedge \psi_2(d_i/t_i)$ is unsatisfiable, for c_i and d_i being fresh constants: if so then we set ψ_1 as $\psi_1(c_i/t_i)$ and ψ_2 as $\psi_2(d_i/t_i)$. Eventually we are left with an unsatisfiable formula $\psi_1 \wedge \psi_2$ where some undesired terms in t_1, \dots, t_n have been removed: the interpolant of ψ_1 and ψ_2 , which can be computed with existing techniques, is likely to be free of the eliminated terms as well. SAFARI retrieves automatically from the input system a list of terms to be abstracted. The terms to abstract are usually set to iterators or variables representing the lengths of the arrays or the bounds of loops. The user can also suggest terms to be added to the list.

Synthesis of Quantified Invariants.

SAFARI is able to generate new quantified variables *if they are needed* to build the safe inductive invariant.

Consider the pseudocode of Fig. 2: the first loop initializes all elements of the array a to 0, while the second loop sets a Boolean flag to *false* if a position with an uninitialized element is found. The program is clearly safe (the assertion is always satisfied), but since the length of the array a is not known, we need

a quantified formula to represent the property *Inv* reached by every execution after the first loop. SAFARI is able to infer that formula automatically, even if the property to check does not contain any quantified variable (see line 9 of Fig. 2 where the property involves the flag f only without any reference to the array a): this process of “synthesis” happens as a consequence of using of existentially quantified labels and term abstraction when refining a spurious (abstract) counterexample. The typical situation is that in which term abstraction succeeds in removing an iterator j from a concrete label of the form $\exists x. (j < n \wedge x = j \wedge a[x] \neq 0)$ to obtain $\exists x. (x < n \wedge a[x] \neq 0)$, where 0 can be any other constant depending on the example. The new label contains no reference to the original iterator j , it is more abstract, and it resembles the structure of *Inv* (once negated: recall that our approach is backward). In short, term abstraction is used during refinement to lift an infeasible concrete trace (corresponding to a spurious abstract counterexample) to the most abstract level with respect to a set of terms. As a side effect, a quantified predicate may be inferred.

Quantifier Handling. The approach behind SAFARI relies on Lazy Abstraction combined with the MCMT framework [1]. Intuitively, during the backward-reachability from the set of error states, we keep track of the array index positions

```

1  i = 0;
2  while( i < n )
3    a[i] = 0;
4    i = i + 1;
5
6  Inv:  $\forall x. (0 \leq x < n) \Rightarrow a[x] = 0$ 
7
8  j = 0; f = true;
9  while( j < n )
10   if( a[j]! = 0 ) f = false;
11   j = j + 1;
12
13 assert ( f );

```

Fig. 2. Pseudo-code for “init and test”

of interest (the positions that are accessed for read) with existentially quantified variables. Safety and covering checks can be performed with dedicated instantiation heuristics. Whereas safety tests are decidable [1], covering tests must be dealt with incomplete algorithms based on clever instantiations (incompleteness of covering tests do not affect the soundness of the tool, they can only affect termination chances). In addition, special care is needed when discovering new predicates via interpolation: quantified queries (expressing trace feasibility) can be Skolemized and instantiated, thus producing equisatisfiable quantifier-free queries. These quantifier-free queries belong to a fragment of the theory of arrays enjoying quantifier-free interpolation. Then, quantifier-free interpolants are computed to refine node labeling, where existential quantifiers are re-introduced by existentially quantifying the Skolem constants (see [1] for details).

Table 1. Total time, calls to SMT, CEGAR iterations, quantified variables in the assertion, size of the covering set, quantified variables in the covering set

Benchmark	Time (s)	SMT-calls	Iter.	P. vars	$ S $	\mathcal{S} vars	Status
binary sort*	0.3	817	2	2	21	4	SAFE
filter (P1)	0.03	27	0	1	2	1	SAFE
filter (P2)	0.04	28	0	1	2	1	SAFE
filter (all)	0.03	34	0	1	3	1	SAFE
find (v1, P1)	0.6	171	3	1	7	5	SAFE
find (v1, P1, buggy)	0.05	71	1	1	-	-	UNSAFE
find (v1, P2)	0.06	65	1	1	4	3	SAFE
find (v1, all)	0.8	246	4	1	12	5	SAFE
find (v2)	0.08	50	1	1	3	1	SAFE
init and test	0.3	352	3	0	13	2	SAFE
initialization	0.1	90	1	1	4	4	SAFE
integers	0.02	20	0	0	2	0	SAFE
max in array	0.9	1237	8	1	29	3	SAFE
max in array (buggy)	0.1	235	2	1	-	-	UNSAFE
partition (v1, P1)	0.05	32	0	1	4	1	SAFE
partition (v1, P2)	0.06	32	0	1	4	1	SAFE
partition (v1, all)	0.08	63	0	1	4	1	SAFE
partition (v2)	0.04	33	0	1	2	2	SAFE
partition (v2, buggy)	0.09	62	0	1	-	-	UNSAFE
selection sort*	0.6	478	4	2	15	3	SAFE
selection sort (buggy)	1.9	1957	8	2	-	-	UNSAFE
strcmp	0.2	308	4	1	12	2	SAFE
strcpy	0.02	16	0	1	2	2	SAFE
vararg*	0.05	48	0	1	3	2	SAFE

4 Experiments

We applied SAFARI to the verification of various problems with arrays. None of these problems can be solved by SAFARI without abstraction. Table 1 reports some experimental results (obtained running SAFARI on an Intel i7 @2.66 GHz, 4GB of RAM running OSX 10.7). More statistics can be found on SAFARI website. The benchmarks have been run with the most efficient options, namely with “Term Abstraction” both for abstraction and refinement. The term abstraction

list is automatically computed for all the benchmarks but those marked with a star: in those few cases a user-defined list has been provided. The table reports variations of the same problems (v1,v2) and properties specified (P1,P2). The benchmarks marked “buggy” were injected with a bug that invalidates the property. To test the flexibility of SAFARI, we also verified some randomly generated problems taken from those shipped with the distribution of the ARMC model-checker (<http://www.mpi-sws.org/~rybal/armc/>). They consists of safety properties of numerical programs without arrays. For those, our tool can solve 23 out of 28 benchmarks with abstraction, but only 9 without using it. For all of these problems, SAFARI automatically retrieves a suitable term abstraction list. For those benchmarks that could be solved even without abstraction, the overhead of abstraction is generally negligible.

Acknowledgements. The work of the first author was supported by the Hasler Foundation under project 09047 and that of the fourth author was partially supported by the “SIAM” project founded by Provincia Autonoma di Trento in the context of the “team 2009 - Incoming” COFUND action of the European Commission (FP7).

References

- Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy Abstraction with Interpolants for Arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012)
- Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic Predicate Abstraction of C Programs. In: PLDI, pp. 203–213 (2001)
- Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
- Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: ICSE, pp. 385–395 (2003)
- Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
- Cousot, P., Cousot, R., Logozzo, F.: A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In: POPL (2011)
- 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)
- Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202 (2002)
- Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. In: IJCAR, pp. 22–29 (2010)
- Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
- Halbwachs, N., Mathias, P.: Discovering Properties about Arrays in Simple Programs. In: PLDI 2008, pp. 339–348 (2008)
- Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: POPL, pp. 58–70 (2002)

13. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
14. Seghir, M.N., Podelski, A., Wies, T.: Abstraction Refinement for Quantified Array Assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
15. Lahiri, S., Bryant, R.: Predicate Abstraction with Indexed Predicates. TOCL 9(1) (2007)
16. Srivastava, S., Gulwani, S.: Program Verification using Templates over Predicate Abstraction. In: PLDI (2009)

BMA: Visual Tool for Modeling and Analyzing Biological Networks

David Benque¹, Sam Bourton¹, Caitlin Cockerton¹, Byron Cook¹,
Jasmin Fisher¹, Samin Ishtiaq¹, Nir Piterman²,
Alex Taylor¹, and Moshe Y. Vardi³

¹ Microsoft Research Cambridge,

² University of Leicester,

³ Rice University

Abstract. BioModel Analyzer (BMA) is a tool for modeling and analyzing biological networks. Designed with a lightweight graphical user interface, the tool facilitates usage for biologists with no previous knowledge in programming or formal methods. The current implementation analyzes systems to establish stabilization. The results of the analysis—whether they be proofs or counterexamples—are represented visually. This paper describes the approach to modeling used in BMA and also notes soon-to-be-released extensions to the tool.

Tool location: <http://biomodelanalyzer.research.microsoft.com/>¹

1 Introduction

In recent years, the verification community has seen a large increase in usage of its tools for modeling and analyzing biological systems (*e.g.* [7–9,13,18], etc.). It is notable, however, that in almost all cases biologists who want to access these tools need a proficiency in computer programming or have to rely on help from computer scientists. The difficulty here is that current tools require sophisticated knowledge of both modelling and analysis techniques, as well as experience with how to combine them. This presents a major barrier to the adoption of formal methods in biology and limits the extent to which its tools can be exploited.

This paper describes BioModel Analyzer (BMA – read “bee-ma”), a graphical tool for the construction and analysis of biological models. The goal of BMA is to support the construction of models using visual notations familiar to specialists in biology, not computer science. More generally, it is intended to illustrate how those with an expertise in biology and an interest in biological modeling, namely biologists, can be given direct access to formal modeling and analysis techniques.

The challenge in this domain is the mismatch between the demands imposed by formal-verification tools and the ways biologists think about and compose models. Formal-verification tools require models to be specified using logical formalisms, whereas biologists tend to see their models in spatial and temporal terms, making distinctions between cells, proteins, genes, etc. and tracing the

¹ Usage requires Microsoft Silverlight to be installed.

pathways between them. For example, to formally verify/analyze graphical models we require that they have precise formal semantics, and yet biologists are, for the most part, unfamiliar with the logical forms giving rise to such formal semantics. Furthermore, the results of formal analysis (*e.g.* abstract counterexamples, invariants, proofs, etc.) are often presented in ways that are unintuitive to biologists. With these issues in mind, BMA has been purposefully designed to fit into biologists' existing ways of understanding and working with biological models, whilst exploiting the benefits of formal-verification techniques. Furthermore, the tool aims to present the results of analysis so that it is intelligible to biologists and facilitates their further scientific investigations.

Related Work. There is a large body of work on usage of formal methods in biological modeling. For example, BioSpice [2] is a repository for many tools and projects. Here, our focus is to support a very specific level of abstraction and somewhat limited, so far, analysis. This allows us to focus on the user interface and connection between formal-verification analysis tools and non-CS experts.

Several attempts have been made to make analysis and modeling tools easier to use for experimental biologists. For example, an English to formal model translator has been proposed [6], as well as a notation describing cells through tables of possible transitions [1]. Graphical notations have also been used for such purposes (*e.g.* [5], [4]). Our approach is similar to that in [5], where a graphical interface is used to create models. However, while their approach requires the biologist to supply state machines that produce the required behavior and programming in either Java or C++ to, *e.g.*, coordinate execution and start it, our aim is to protect the non-CS expert from the need in such knowledge. In our approach the user supplies the rules that govern the behavior and induce the state machines. The distinguishing characteristics of BMA are a) its high-performance analysis engine [3] and b) its focus on improving the interface between the biologist and the formal-verification/analysis tool.

Motivated by SBML [10], BMA supports output to a custom XML format. Thus, it is possible to interface other analysis tools with BMA models.

2 Designing Biological Networks

BMA focuses on a specific domain of modeling. Our models are composed of one or more cells, cell elements (*i.e.*, proteins), and connections that specify the relations between these elements. These elements represent the biological components very abstractly and at high level. As mentioned, BMA is intended to support the design of models using graphical notations. This notation has been intentionally designed to be familiar to biologists and match the representations they commonly use in modeling (*e.g.*, [11,16,19]). These graphical models produce an underlying semantic layer based on Qualitative Networks (QN) [15]. Thus, a drawing gives rise to a model that can be automatically analyzed. All advanced features of QNs are available to advanced users.

Fig. 1 shows a simple model. On the canvas there is an isolated protein and two cell membranes, containing proteins. Each cell has two receptors (in green) and

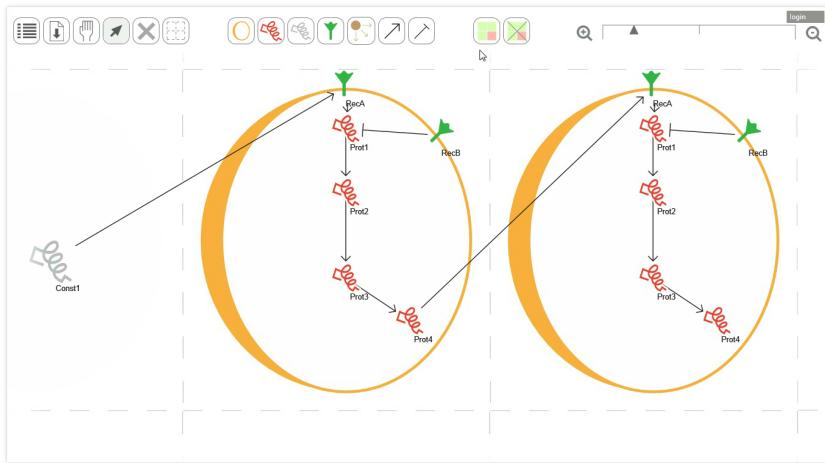


Fig. 1. A small model including two cells, receptors (green shape on the membranes), cell signalling (red coils), and a protein in the outer cell surrounding (grey coil)

the arrows show a cascade that flows from the cell on the left to the cell on the right. The arrows and bar-arrows are the usual notation to indicate activation (positive influence) and inhibition (negative influence) between proteins. In QNs, these relationships are translated into rules that govern the behavior of proteins based on the values level of activity of proteins affecting them.

The user constructs the models by dragging and dropping elements onto a gridded canvas. Cell membranes that are added fill an entire square in the grid. These have no functional use in the analysis and simply allow the user to pictorially represent a cell. Proteins can be placed either in or outside of these membranes. They can be represented as either receptors, that lie on the cell membrane, or as stand-alone proteins in or outside cells. Connections in the form of arrows or bar-arrows can be drawn between proteins.

Each protein can be named and given a finite value range to represent the protein's concentration level. For example, a range of [0..3] for a protein might signify concentration levels “off”, “low”, “medium”, or “high”. In Fig. 2 we see the additional window in which the name and value ranges are set.

Semantically, a graphical BMA model translates to a transition system, which can be analyzed using existing formal analysis tools. BMA’s current underlying analysis is designed to prove stabilization [20]. A model is stabilizing if in every execution all variables (*i.e.*, protein value ranges) eventually reach a single fixed value and there is no possibility of further change. In addition, all executions result in the same fixed value for each protein. The user can execute this analysis and check whether the resulting model is stabilizing by pressing the “proof” button in the tool. Support of more temporal properties is currently developed.

The output of the test for stabilization is displayed graphically. Proteins that result in a fixed value are colored green and annotated by their value. Proteins that do not reach a fixed value are colored red and annotated by their range of

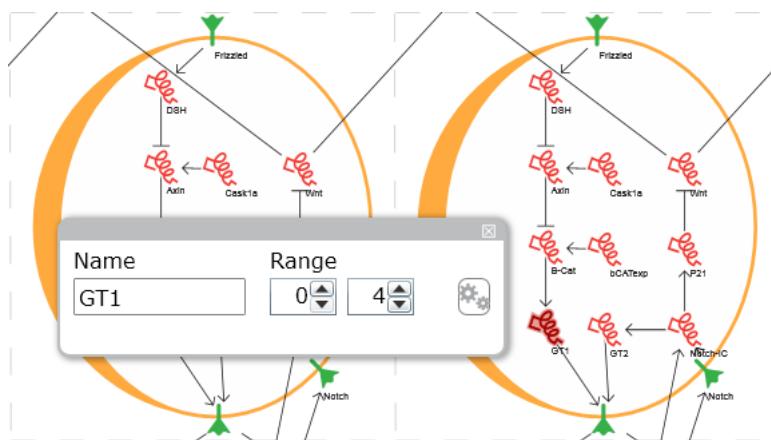


Fig. 2. Partial view of a more complex model and the window in which the protein name and value ranges are entered

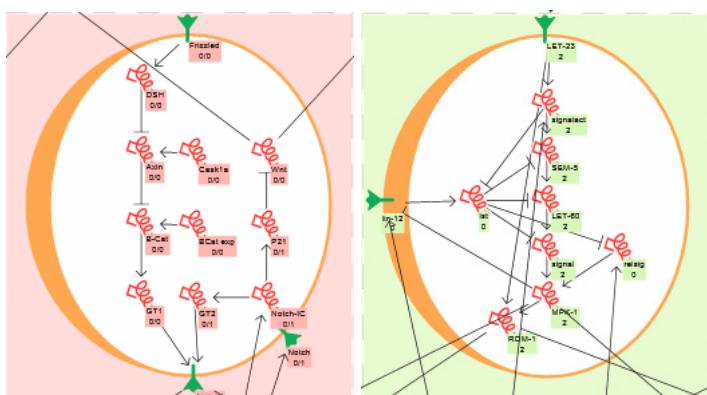


Fig. 3. Cell from non-stabilizing model (left) and cell from stabilizing model (right)

possible values. With models that fail to stabilize, the user can access details of the analysis and the individual steps executed by right-clicking on the proof button and selecting the appropriate contextual menu option. Example results from stability analysis are given in Fig. 3. The left image shows a cell from a non-stabilizing model and the right a cell from a stabilizing model.

As sometimes happens with model checking, when properties are proven for a system, the reasons are not always clear. In biology, and with BMA, especially, this is problematic because why and how stabilization occurs have important implications for what biologists might do next in their research. To help users understand why stabilization holds, we are currently working on an animated visual representation of the proof's execution.

3 Analyzing Biological Networks

As we have noted, the graphical models users produce are formally represented using Qualitative Networks (QN) [15]. The tool automatically translates the graphical models to a QNs. The QNs include variables representing the concentration of proteins as a discrete value in a fixed range. Values of variables change gradually according to interactions between the proteins. QNs and similar formalisms (*e.g.*, genetic regulatory networks [17]) are simple enough to be represented graphically and expressive enough to capture interesting biological phenomena (*e.g.*, [2,14,15,17]).

Qualitative Networks. A *qualitative network* (QN) is $Q = (V, T, N)$, where $V = (v_1, v_2, \dots, v_n)$ is a set of variables ranging over $\{0, 1, \dots, N\}$ and $T = (T_1, \dots, T_n)$ are their respective target functions. A state of the system is an assignment $s : V \rightarrow \{0, 1, \dots, N\}$. Let Σ denote the set of all possible states. A *target function* $T_i \in T$ is $T_i : \Sigma \rightarrow \{0, 1, \dots, N\}$. Intuitively, in a given state s , variable v_i “would like” to get the value $T_i(s)$. However, values of variables change by at most 1. The *successor* of state s is s' , where for every $v_i \in V$ we have:

$$s'(v_i) = \begin{cases} s(v_i) + 1 & s(v_i) < T_i(s), \\ s(v_i) & s(v_i) = T_i(s), \\ s(v_i) - 1 & s(v_i) > T_i(s). \end{cases} \quad (1)$$

Thus, a QN defines a transition system over Σ . All variables change their value synchronously by following their target functions. We abuse notation and write also $T : \Sigma \rightarrow \Sigma$ as the function that associates with a state s its successor s' .

Each protein in the design corresponds to a variable in the underlying Qualitative Network. Perhaps the most complex feature of QNs is the target functions. In order to enable novice users to bypass the need to define target functions, we set a default target function induced by the activations and inhibitions applied to a given protein. Denoted in short as $ave(pos) - ave(neg)$. That is, the weighted average of the proteins that activate the protein minus the weighted average of the proteins that inhibit the protein.

Custom Target Functions. In the graphical representation, users can customize the target function by clicking on the cog-wheel in a protein’s property window (see Fig. 2). Target functions are defined using a small language of possible mathematical operations (*e.g.* $+$, $-$, $/$ max , etc). For example we might use $T_i = N - \frac{v_1 + v_2}{2}$, which models a situation where v_1 and v_2 affect v_i negatively. With no negative influence (*i.e.* when the values of v_1 and v_2 are 0), v_i aims to settle in its maximal value. When v_1 and v_2 are high, v_i aims to decrease. This models a situation in which a protein is constitutively produced unless proteins v_1 and v_2 inhibit its production. A more complex target function is $T_i = max(0, min(2 - v_1, 1)) \times v_2$. In this case if v_1 is 0 or 1 then v_i aims to follow v_2 . However, if v_1 is more than 1, then v_i aims to decrease to 0. This models a situation when above a certain threshold protein v_1 strongly inhibits v_i but otherwise protein v_2 positively influences v_i .

Stabilization. We say that a state s is *recurring* if it is possible to reach s from itself after a finite number of applications of T . That is, for some $i \geq 1$ we have $s = T^i(s)$. We note that the number of states of a QN is finite, hence, the set of recurring states cannot be empty. A QN is *stabilizing* if for some state s we have $s = T(s)$ and no other state is recurring.

BMA attempts to prove stabilization using the approach from [3] which combines a search for thread-modular proofs of liveness together with techniques from abstract interpretation and the intervals domain.

4 Conclusion

In this paper we introduce a new graphical tool for modeling and analyzing biological networks. The tool is intended for use by biologists. The current implementation analyzes systems to establish stabilization, with support for additional temporal properties under development. The results of BMA's analysis are represented visually: counterexamples are displayed by showing regions of the network that could take on additional values. In the future, proofs will be demonstrated by visually displaying the lemmas found during the proof search. We are currently working on introducing additional types of analysis. On the one hand, introducing analysis that does not require the user to specify logical formulas. On the other hand, finding intuitive (and graphical) ways to allow users to specify logical formulas and feeding back the output of the analysis.

References

1. Amir-Kroll, H., Sadot, A., Cohen, I., Harel, D.: GemCell: A generic platform for modeling multi-cellular biological systems. *TCS* 391(3) (2008)
2. Beyer, A., Thomason, P., Li, X., Scott, J., Fisher, J.: Mechanistic insights into metabolic disturbance during type-2 diabetes and obesity using qualitative networks. *TCSB* 12 (2010)
3. Cook, B., Fisher, J., Krepka, E., Piterman, N.: Proving Stabilization of Biological Systems. In: Jhala, R., Schmidt, D. (eds.) *VMCAI* 2011. LNCS, vol. 6538, pp. 134–149. Springer, Heidelberg (2011)
4. Danos, D., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-based modelling and model perturbation. *TCSB* 11 (2009)
5. Efroni, S., Harel, D., Cohen, I.R.: Toward rigorous comprehension of biological complexity: modeling, execution, and visualization of thymic T-cell maturation. *Genome Res.* 13(11) (2003)
6. Errampalli, D.D., Priami, C., Quaglia, P.: A formal language for computational systems biology. *Omics* 8(4) (2004)
7. Fisher, J., Piterman, N., Hajnal, A., Henzinger, T.: Predictive modeling of signaling crosstalk during *C. elegans* development. *PLoS Comp. Bio.* 3(5) (2007)
8. Ghosh, R., Tiwari, A., Tomlin, C.: Automated Symbolic Reachability Analysis; with Application to Delta-Notch Signaling Automata. In: Maler, O., Pnueli, A. (eds.) *HSCC* 2003. LNCS, vol. 2623, pp. 233–248. Springer, Heidelberg (2003)
9. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic Model Checking of Complex Biological Pathways. In: Priami, C. (ed.) *CMSB* 2006. LNCS (LNBI), vol. 4210, pp. 32–47. Springer, Heidelberg (2006)
10. Hucka, M., Finney, A., Sauro, H., Bolouri, H., Doyle, J., Kitano, H.: The systems biology markup language (SBML). *Bioinfo.* 19(4), 524–531 (2003)

11. Klerkx, E., Alarcón, P., Waters, K., Reinke, V., Sternberg, P., Askjaer, P.: Protein kinase vrk-1 regulates cell invasion and egl-17/fgf signaling in *C. elegans*. *Dev. Bio.* 335(1) (2009)
12. Kumar, S., Feidler, J.: BioSPICE: A computational infrastructure for integrative biology. *OMICS* 7(3), 225 (2003)
13. Li, C., Nagasaki, M., Ueno, K., Miyano, S.: Simulation-based model checking approach to cell fate specification during *C. elegans* vulval development by hybrid functional petri net with extension. *BMC Sys. Bio.* 3(42) (2009)
14. Sanchez, L., Thieffry, D.: Segmenting the fly embryo: a logical analysis fo the pair-rule cross-regulatory module. *J. of Theo. Bio.* 244 (2003)
15. Schaub, M., Henzinger, T., Fisher, J.: Qualitative networks: A symbolic approach to analyze biological signaling networks. *BMC Sys. Bio.* 1(4) (2007)
16. Sundaram, M.V.: The love-hate relationship between Ras and Notch. *Genes Dev.* 19(16) (2005)
17. Thomas, R., Thieffry, D., Kaufman, M.: Dynamical behaviour of biological regulatory networks—i. biological role of feedback loops and practical use of the concept of the loop-characteristic state. *Bull. of Math. Bio.* 55(2) (1995)
18. Wang, D., Cardelli, L., Phillips, A., Piterman, N., Fisher, J.: Computational modeling of the egfr network elucidates control mechanisms regulating signal dynamics. *BMC Sys. Bio.* 3(1), 118 (2009)
19. Yoo, A.S., Bais, C., Greenwald, I.: Crosstalk between the EGFR and LIN-12/Notch pathways in *C. elegans* vulval development. *Science* 303(5658) (2004)
20. Zotin, A.: The Stable state of organisms in thermodynamic bases of biological processes: Physiological Reactions and Adaptations (1990)

APEX: An Analyzer for Open Probabilistic Programs*

Stefan Kiefer¹, Andrzej S. Murawski², Joël Ouaknine¹,
Björn Wachter¹, and James Worrell¹

¹ Department of Computer Science, University of Oxford, UK

² Department of Computer Science, University of Leicester, UK

Abstract. We present APEX, a tool for analysing probabilistic programs that are open, i.e. where variables or even functions can be left unspecified. APEX transforms a program into an automaton that captures the program’s probabilistic behaviour under all instantiations of the unspecified components. The translation is compositional and effectively leverages state reduction techniques. APEX can then further analyse the produced automata; in particular, it can check two automata for equivalence which translates to equivalence of the corresponding programs under all environments. In this way, APEX can verify a broad range of anonymity and termination properties of randomised protocols and other open programs, sometimes with an exponential speed-up over competing state-of-the-art approaches.

1 Introduction

APEX is an analysis tool for *open* probabilistic programs. Such programs are very well suited to analyse randomised algorithms and, in particular, anonymity in security protocols: they (i) represent the behaviour of an algorithm succinctly for a range of inputs, and (ii) allow to differentiate internal behaviour from externally observable behaviour, so that anonymity can be established by proving that secret information is not externally observable.

APEX’s key technology is the use of *game semantics* [2], which provides a compositional translation of open probabilistic programs to *probabilistic automata*. Probabilistic automata [10] are essentially nondeterministic automata whose transitions are decorated with probabilities. A theorem [8] guarantees that two open probabilistic programs are equivalent if and only if the probabilistic automata are *language equivalent*, i.e., accept every word with the same probability. Language equivalence between probabilistic automata reduces to a linear-algebra problem for which efficient algorithms have been developed, see [5] and the references therein. APEX performs both the translation from programs to automata and the language-equivalence check. Thus, given two open probabilistic programs, APEX either proves them equivalent, or provides a word that separates the programs.

APEX has been applied to a range of case studies [6]: it provides the most efficient automatic verification of dining cryptographers to date, an analysis of

* Research supported by EPSRC. The first author is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

Hibbard's algorithm for random tree insertion, and of Herman's self-stabilisation algorithm.

Example: The Grade Protocol in APEX. We illustrate the use of APEX by analysing a protocol for a group of students, who have been graded and would like to find out the sum of their grades (e.g., to compute the average) without revealing the individual grades. This is accomplished with the following randomised algorithm. Let $S \in \mathbb{N}$ be the number of students, and let $\{0, \dots, G - 1\}$ ($G \in \mathbb{N}$) be the set of grades. Define $N = (G - 1) \cdot S + 1$. We assume that the students are arranged in a ring, as depicted in Figure 1, and that each pair of adjacent students shares a random integer between 0 and $N - 1$. Thus a student shares a number l with the student on its left and a number r with the student on its right, respectively. Denoting the student's grade by g , the student announces the number $(g + l - r) \bmod N$. The sum of the announced numbers ($\bmod N$) is telescoping, so it equals the sum of all grades. We require that no participant glean anything from the announcements other than the sum of all grades. This correctness condition can be formalised by a specification in which the students make random announcements subject to the condition that the sum of the announcements equals the sum of their grades.

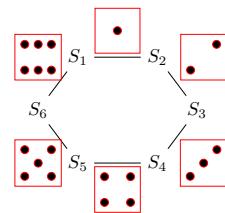


Fig. 1.

<pre>\\" Implementation const N := S*(G-1)+1; grade:int%G, out:var%N - var%(S+1) i; i:=0; var%N first; first:=rand[N]; var%N r; r:=first; while(i<S) do { var%N l; i:=succ(i); if(i=S) then left:=first else left:=rand[N]; out:=(grade + 1) - r; r:= 1; } : com</pre>	<pre>\\" Specification const N := S * (G-1) + 1; grade:int%G, out:var%N - var%N i; var%N total; i:=1; while(i) do { total := grade + total; var%N r; r := rand[N]; out:=r; total := total - r; i:=succ(i) }; out:= grade + total: com</pre>
--	---

Fig. 2. Grade protocol: APEX programs

Figure 2 shows two APEX programs: on the left, the implementation of the grade protocol to be verified against the specification program, given on the right. The input language of APEX is an imperative sequential programming language with a C-like syntax and support for procedures and arrays. There are several constructs to define probability distributions, e.g., the expression `rand[N]` gives

a uniform distribution over the numbers $\{0, \dots, N - 1\}$, `coin` is a shorthand for `rand[2]`, and `coin[0:1/4, 1:3/4]` a biased coin. Variables are defined over finite ranges, e.g., `var%N total;` declares a variable over range $\{0, \dots, N - 1\}$. There are internal variables such as the counter `i` which are defined locally, and externally observable program variables through which the program communicates with the outside world. Externally observable variables are declared before the turnstile symbol `|-`; in our example, there are two such variables: the individual grades of students are read from variable `grade`, while `out` is an output variable that announces the random numbers generated by the protocol.

APEX checks whether the programs are equivalent. In this case it finds that they are, so one can conclude that the grade protocol guarantees anonymity.

2 How APEX Works

APEX is implemented in C and OCaml and consists of approximately 18K lines of code. Figure 3 shows APEX’s architecture. It has two main components: an automaton construction routine and an equivalence checker.

2.1 Automaton Construction

The automaton constructor builds a probabilistic automaton using game semantics. The construction works at the level of the program’s abstract syntax tree (AST). The leaves of the tree correspond to variables and constants, while internal nodes correspond to semantic operations of the language like arithmetic expressions, probabilistic choice, conditionals, sequential composition of commands, and loops. APEX labels each AST node with the automaton that captures its semantics, by proceeding bottom-up and composing the automata of the children. Ultimately, the automaton computed for the root of the AST gives the semantics of the entire program.

Figure 4 shows the probabilistic automata obtained from the programs in Figure 2 (with $S = 2$ and $G = 2$). Transitions in the automaton contain only reads and writes to *observable* variables; e.g., label `1.grade` means that value 1 has been read from variable `grade` and `write(2).out` means that value 2 has been written to variable `out`. Actions on *internal* variables are hidden. Each transition is also labeled (comma-separated) with a probability.

2.2 Equivalence Checking

To check equivalence of the input programs, it suffices for APEX to check the corresponding probabilistic automata for language equivalence. If they are not

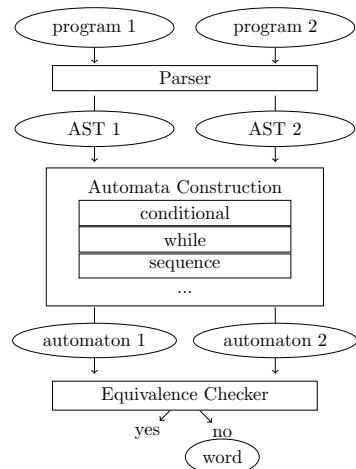


Fig. 3. APEX architecture

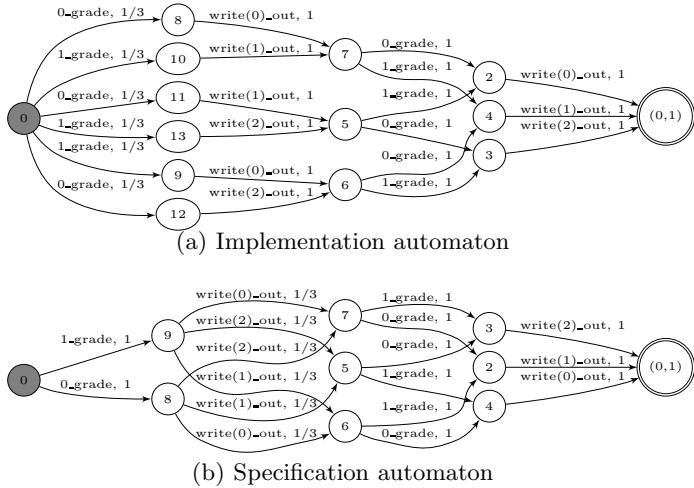


Fig. 4. Probabilistic automata for the grade protocol

equivalent, APEX presents a counterexample word, which corresponds to a run which is executed with different probabilities in the two programs. APEX uses efficient linear-algebra-based algorithms [5] for the equivalence check and the counterexample extraction.

3 Comparison with Other Tools

PRISM [4] is the leading probabilistic model checker with a large collection of case studies. As regards anonymity, model checking is considerably less convenient than equivalence checking. In [9], for example, the authors establish anonymity of the Dining Cryptographers protocol by considering all possible visible behaviours, and proving for each that the likelihood of its occurrence is the same regardless of the payer. This leads to exponentially large specifications, and correspondingly intractable model-checking tasks.¹ In practice, a proper verification of the protocol can only be carried out for a handful of cryptographers [6], while APEX scales to around 800 cryptographers on a state-of-the-art workstation.

Mage [1] is a software verification tool based on game semantics which applies to *non-probabilistic* programs. To our knowledge, APEX is the only game-semantics-based tool for probabilistic programs.

4 Novel Features

We discuss several new features and algorithmic improvements that have not been covered in previous publications [5,6,8].

¹ The state space of the underlying Markov chain generated by PRISM also grows exponentially, but this is mitigated by PRISM's use of symbolic representations in the form of MTBDDs.

Variable Binding by Reachability Analysis. Game semantics views a state variable as an automaton that answers read requests and stores values on writes. The subprogram in which the variable lives, i.e. its scope, is an open program in terms of this variable. A variable-binding operator closes this subprogram by synchronous composition of the subprogram with the automaton of the variable. At the semantic level, both variable and scope are represented as automata.

The automata semantics of the binding operator is defined in terms of the synchronous product of the two automata, in which transitions involving reads and writes of the variable are turned into silent ε -transitions which are to be removed e.g. using ε -removal algorithms [7] to obtain the ultimate result.

In the previous implementation of APEX, the product automaton was formed by copying the automaton that represents the scope of the variable k times where k is the number of potential values of the variable (which coincides with the number of states of the automaton that represents the variable). In practice, many of the state-value pairs thus created turn out to be unreachable because the variable typically only takes on a subset of the potential values in its domain at a certain program location. In the new version of APEX, the binding operator computes the automaton by enumerating reachable state-value pairs only. In this way, the peak number of states of the constructed automata is reduced.

Live-Variable Detection. The reachability analysis for variable binding can be further optimised by taking into account liveness information. A variable is live if there is a path to a usage of the variable, and is dead otherwise. At a state in which a variable is dead, the product construction can lump all state-value pairs with the same state, as the value of the variable has become irrelevant.

Liveness analysis is collected by a simple procedure that tracks the liveness of the bound variable with a bit and proceeds backwards from the reading occurrences of the variable in an automaton. This information is subsequently used in the product construction.

Early ε -Removal. In the previous version of APEX, product construction and ε -removal were two distinct phases. Now linear chains of ε -transitions are immediately removed in the product construction, while branching and cyclical structures are left to the full ε -removal routine. By eliminating the ‘simple’ cases of ε -transitions, subsequent steps such as bisimulation and ε -removal run faster.

Lumping Bisimulation. APEX applies lumping bisimulation to reduce automata size. The bisimulation routine runs frequently during the compositional automata construction. Hence its performance is crucial. Recently we have improved the underlying algorithms. APEX now features a signature-based refinement algorithm [3] that computes a strong bisimulation. Key to its efficiency is to leverage very inexpensive algorithms that compute a coarse pre-partition to which the precise signature-based partition refinement is subsequently applied to obtain the final lumping. Pre-partitioning proceeds in two phases: (1) APEX lumps states according to their minimal distance from an accepting state; whereby the distance is computed by a backwards depth-first search from the accepting states. (2) APEX

runs an approximate version of signature-based refinement which utilises hash values of signatures to refine the partition, instead of comparing signatures with each other. Both steps have helped to significantly lower the cost of lumping.

Counterexample generation. The previous version of APEX did not provide diagnostic information in case two programs were inequivalent. Now APEX generates a counterexample word, i.e., a word which the programs accept with a different probabilities. The new feature is enabled by the techniques presented in [5].

Online Tool Demo. We have implemented an online version of APEX which offers a convenient user interface and runs on any device with a recent web-browser:

www.cs.ox.ac.uk/apex

The user can either select from existing case studies, load case studies on the server, or drag & drop into the input window. Automata are displayed as scalable vector graphics (SVG). The view can be zoomed with the mouse wheel and the viewing window can be moved by panning. Further the interface has an equivalence checking mode in which counterexamples are shown. Internally, the tool runs on a server and the dynamic web pages through which the user interacts with APEX are generated by PHP scripts.

References

1. Bakewell, A., Ghica, D.R.: On-the-Fly Techniques for Game-Based Software Model Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 78–92. Springer, Heidelberg (2008)
2. Danos, V., Harmer, R.: Probabilistic game semantics. ACM Transactions on Computational Logic 3(3), 359–382 (2002)
3. Derisavi, S.: Signature-based symbolic algorithm for optimal markov chain lumping. In: QEST, pp. 141–150 (2007)
4. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A Tool for Automatic Verification of Probabilistic Systems. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
5. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: Language Equivalence for Probabilistic Automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 526–540. Springer, Heidelberg (2011)
6. Legay, A., Murawski, A.S., Ouaknine, J., Worrell, J.: On Automated Verification of Probabilistic Programs. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 173–187. Springer, Heidelberg (2008)
7. Mohri, M.: Generic e-removal and input e-normalization algorithms for weighted transducers. Int. J. Found. Comput. Sci. 13(1), 129–143 (2002)
8. Murawski, A.S., Ouaknine, J.: On Probabilistic Program Equivalence and Refinement. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 156–170. Springer, Heidelberg (2005)
9. PRISM case study: Dining Cryptographers,
www.prismmodelchecker.org/casestudies/dining_crypt.php
10. Rabin, M.O.: Probabilistic automata. Information and Control 6 (3), 230–245 (1963)

Recent Developments in FDR*

Philip Armstrong, Michael Goldsmith, Gavin Lowe, Joël Ouaknine,
Hristina Palikareva, A.W. Roscoe, and James Worrell

Department of Computer Science, Oxford University, UK
`{phila,michael,gavinl,joel,hrip,awr,jbw}@cs.ox.ac.uk`

Abstract. We describe and report upon various substantial extensions of the CSP refinement checker FDR including (i) the direct ability to handle real-time processes; (ii) the incorporation of bounded model checking technology; (iii) the development of conservative and highly efficient static analysis algorithms for guaranteeing livelock-freedom; and (iv) the development of automated CEGAR technology.

1 Introduction

FDR, standing for *Failures Divergence Refinement*, is the best-known tool supporting CSP [13, 21, 22]. It was originally released in 1991/2 and underwent a major re-write in 1994/5 to become FDR2. All subsequent developments have extended FDR2, including those reported in this paper. It is extensively described in [21, 22]. Most of its functionality is based around proving or refuting refinement between finite-state CSP processes, where refinement is over a selection of semantic models with different expressive powers. The best known of these are *traces*, which proves partial correctness, *failures*, which additionally handles deadlock, and *failures-divergences*, which captures a wide variety of total correctness properties. *Refinement* over one of these models is always reverse containment between the sets of relevant behaviours representing the processes.

Traditionally it has been an explicit model checker—capable of exploring millions of states per minute on a modern workstation—supported by state-space compression techniques and the partial-order compression *chase*, as described in [21, 22].

FDR has been widely used in research, teaching and industry [3, 6, 15], and is well known for its use in security analysis [14]. Until 2007 it was a product of Formal Systems (Europe) Ltd, but since 2008 it has been developed in Oxford University under support from EPSRC, ONR and industry. The present paper summarises the new features added in this latter phase.

In recent years FDR has been used as the back-end of verification engines aimed at notations other than CSP. Casper (security) and the shared-variable analyser SVA (see Chapters 18 and 19 of [22]) are examples, as well as a number of proprietary industrial tools.

* Research supported by EPSRC, ONR, Verum BV and Qinetiq.

2 New Techniques and Features

Modelling and Verifying Real-Time Systems. Two different timed extensions have been developed for CSP. The first is Timed CSP [20], which is a real-time interpretation of Hoare’s CSP notation [13]. In its usual form it adds a single construct to CSP, namely $\text{WAIT } t$ which waits t units of time before terminating successfully. It is possible to express a wide variety of time-based operations such as time-outs in terms of $\text{WAIT } t$ and standard CSP. Timed CSP generally assumes a continuous clock in the hands of external observers. A second form, *tock*-CSP, was introduced by Roscoe [21] as a medium for verifying discretely timed systems on FDR, most naturally with an internal clock signal. A special event *tock* represents the regular passage of time.

Thanks to the idea of *digitisation* introduced by Henzinger, Manna and Pnueli [12] and developed for Timed CSP by Ouaknine [17, 18], it is possible to establish theoretical connections among continuous Timed CSP, the Timed CSP language with a discrete semantics, and *tock*-CSP. It is therefore possible to model check Timed CSP programs, drawing conclusions about their continuous semantics by a relatively modest modification to FDR along the lines suggested in [17, 22]. This modification—described in more detail in [4]—takes the form of a mode within FDR that instructs it to interpret the syntax inside it as a Timed CSP process and translate it into semantically equivalent *tock*-CSP. It is possible, and frequently very useful, to mix Timed CSP, *tock*-CSP and ordinary CSP in the same script and indeed in the same process or refinement check.

We can check Timed CSP processes for refinement against specifications formulated in Timed CSP itself, in *tock*-CSP, or—when time is suitably abstracted—in ordinary CSP. As reported in [23], it is possible to check Timed CSP for noninterference (i.e. information flow) properties and therefore find or prove the absence of *timing channels* that pass information between mutual users of a system.

This Timed CSP mode of FDR is a recent development and we have not yet had time to try it on serious industrial examples. We have, however, tried it on several well known benchmarks such as Fischer’s mutual exclusion protocol and the puzzle in which a number of soldiers have to cross a bridge in pairs using a torch. In both of these it demonstrated great efficiency: Table II shows results for the first of these in comparison to UPPAAL [2] and PAT [1]. Further results can be found in [4].

Bounded Model Checking and Temporal k -Induction. For the traces model of CSP, FDR now supports an alternative refinement engine employing symbolic techniques based on Boolean satisfiability (SAT). In particular, FDR features bounded model checking (BMC) [7], that can be used for bug detection, and temporal k -induction [11], which builds upon BMC, aims at establishing inductiveness of properties and is capable of both bug finding and establishing the correctness of systems. The symbolic engine [19] adopts FDR’s implicit operational representation based on supercombinators [22], but explores this using

Table 1. Timed CSP. Times reported are in seconds, with \star denoting memout. Comparison against UPPAAL 4.0.13 and PAT 3.40. The columns titled PAT-zone and PAT-digit denote, respectively, PAT using zone abstraction and digitisation as underlying engines. All experiments were performed on a 2.6 GHz PC with 2 GB RAM running Linux Fedora.

Benchmark	FDR	Uppaal	PAT-zone	PAT-digit
Fischer mutual exclusion-6	0	0	13	7
Fischer mutual exclusion-7	0	0	196	85
Fischer mutual exclusion-8	0	2	\star	\star
Fischer mutual exclusion-10	2	20	\star	\star
Fischer mutual exclusion-12	18	312	\star	\star

SAT rather than explicitly. For both BMC and k -induction, FDR offers configurable support for a SAT solver (MiniSAT, PicoSAT or ZChaff, all used in incremental mode), Boolean encoding (one-hot or binary), traversal mode (forward or backward), etc. The BMC engine sometimes substantially outperforms the original explicit state-space exploration, especially for complex tightly-coupled combinatorial problems, as reported in [19]. For k -induction, the completeness threshold blows up in all cases, due to concurrency, and, therefore, high performance depends on whether or not the property is k -inductive for some small value of k . Thus we have only seen SAT outperform FDR when there are counterexamples.

Static Analysis for Establishing Livelock Freedom. FDR now supports an alternative back end for establishing livelock freedom. Livelock, also called divergence, indicates that a process is unresponsive due to being engaged forever in internal computations. The new back-end relies on static analysis of the syntactic structure of a process rather than explicit state exploration. It employs a collection of rules to calculate a sound approximation of the fair/co-fair sets of events of a process [16]. The rules either safely classify processes as livelock-free or report inconclusiveness, thereby trading accuracy for speed. The algorithms generate and manipulate various sets of events in a fully symbolic way. The choice of an underlying symbolic engine is configurable, with support for using a SAT engine (based on MiniSAT 2.0), a BDD engine (based on CUDD 2.4.2), or running a SAT and a BDD analyser in parallel and reporting the results of the first one to finish. Experiments indicate that the static analyser is substantially more efficient than the exhaustive-search approach, outperforming it by multiple orders of magnitude whilst exhibiting a low rate of inconclusive results. We experimented with a wide range of benchmarks, including parameterised, parallelised, and piped versions of Milner’s Scheduler, the Alternating Bit Protocol, the Sliding Window Protocol, the Dining Philosophers, Yantchev’s Mad Postman Algorithm, etc., as reported in [16].

CEGAR. We developed abstraction/refinement schemes for the traces, failures and failures-divergences models and embedded them into a fully automated and compositional counterexample-guided abstraction refinement framework (CEGAR) [10]. An initially coarse abstraction of the system is iteratively refined (i.e. made more precise) on the basis of spurious counterexamples until either a genuine counterexample is derived or the property is proven to hold. We exploit the compositionality of CSP for the stages of initial abstraction, counterexample validation and abstraction refinement, extending the framework proposed in [9, 8]. Generally, we adopt lazy refinement strategies that yield coarser abstractions even though it takes a greater number of iterations to converge. Experiments can show performance enhancement when verifying both safety and liveness properties, as illustrated in Table 2.

Table 2. CEGAR. Times reported are in seconds, with * denoting a 30-minute timeout. The last column titled # reports the number of iterations that it takes for CEGAR to converge. All experiments were performed on a 3.07GHz Intel Xeon processor with 8 GB RAM running Linux Ubuntu.

Property	Benchmark	FDR	CEGAR	#
(safety), holds	Milner-10	0	0.03	21
	Milner-20	158	0.07	41
	Milner-30	*	0.16	61
	Milner-100	*	4.42	201
	Milner-200	*	40.01	401
(liveness), holds	Deadlock Postman-3	4	0.03	4
	Mad Postman-5	*	0.22	4
	Mad Postman-7	*	1.49	4
	Mad Postman-9	*	7.13	4

The columns for FDR represent its use with none of its compression functions used. Compression can be used effectively on these systems, but of course requires skill in picking the right compression and compression strategy.

New Semantic Models of CSP. While traces, failures and failures-divergences remain the most generally used models in FDR, it is sometimes useful to have the expressive power of richer models. FDR now supports the *revivals* and *refusal testing* models [22], together with their divergence-strict analogues. We eventually hope to support almost the full range of models reported in Chapters 10–12 of [22], including some which support non-strict reasoning about divergence. Compositional reasoning about Timed CSP and priority each require one of these stronger models.

Divergence-Respecting Weak Bisimulation. The range of compression operators is now augmented with divergence-respecting weak bisimulation (DRWB):

the largest weak bisimulation which does not identify any immediately divergent node with one that is not. Typically, DRWB does not achieve quite the same degree of compression as the combination of strong bisimulation and *diamond* compression, which is frequently used with FDR. However, it has the great advantage that it is faithful to all CSP models and also inside the priority operator, something that is not true of diamond compression (which works for traces and failures based models only). DRWB compression was, for example, crucial to the efficiency of the timed noninterference work described above.

A Priority Operator. In [22] Roscoe proposed a priority operator for CSP that would fit within the general operational semantic framework of CSP and for which the most refined of the abstract semantic models described in that book would be compositional. It is a generalisation of the *timing priority* model that had been used for some time with *tock*-CSP models of timed systems, as discussed above. It is an operator which takes a CSP process in which no actions are intrinsically prioritised and returns another one of the same type. In theory, we can take any partial order on the actions of an operational semantics: ordinary visible actions together with the internal τ and termination signal \checkmark , in which the latter two are both maximal. The priority operator then blocks any action at an operational state when that state has one of higher priority. We support orders in which there are a number of distinct priority levels—sets of equal-priority events—that are linearly ordered, with the first of these sets of events the possibly empty set at the same priority level as $\{\tau, \checkmark\}$. These levels need not partition the entire alphabet, with any events outside their union neither blocking nor being blocked by any other. Thus $\text{prioritise}(P, \{\}, \{\text{tock}\})$ represents the operator that gives τ and \checkmark higher priority than *tock*, with no other priorities enforced.

This operator was implemented thanks to industrial funding from Verum after it was discovered [5] that priority plus other CSP operators such as renaming could be used to determine whether a system can still diverge when we disallow some infinite τ sequences in which hidden events from a set M are infinitely often accompanied by offers of events from some set A . This was required for important availability checks in Verum’s ASD tool [6], which has FDR embedded as its verification engine.

3 Technical Details, Availability and Usage

FDR is largely written in C++ and runs on Linux, Mac OS X and Solaris on SPARC. The binaries, as well as a user manual, are available for download from:

<http://www.cs.ox.ac.uk/projects/concurrency-tools/>

There are two ways of using FDR: either through its own GUI or through a command-line interface that is primarily used by other verification tools which use FDR as a back end. Details can be found in the user manual. Collections of CSP scripts can be downloaded from <http://www.cs.ox.ac.uk/ucs/CSPM>.

References

- [1] PAT: Process analysis toolkit, <http://www.comp.nus.edu.sg/~pat/>
- [2] UPPAAL, <http://www.uppaal.org/>
- [3] Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.): Communicating Sequential Processes. LNCS, vol. 3525. Springer, Heidelberg (2005)
- [4] Armstrong, P., Lowe, G., Ouaknine, J., Roscoe, A.W.: Model checking Timed CSP. In: HOWARD. EasyChair, pub. (to appear, 2012)
- [5] Armstrong, P., Hopcroft, P.J., Roscoe, A.W.: Fairness checking through priority (to appear, 2012)
- [6] Broadfoot, G.H., Hopcroft, P.J.: A paradigm shift in software development. In: Proceedings of Embedded World Conference, Nurmemburg (2012)
- [7] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
- [8] Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing 17(4), 461–483 (2005)
- [9] Chaki, S., Ouaknine, J., Yorav, K., Clarke, E.M.: Automated compositional abstraction refinement for concurrent C programs: A two-level approach. Electronic Notes in Theoretical Computer Science, vol. 89 (2003)
- [10] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
- [11] Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science, vol. 89 (2003)
- [12] Henzinger, T.A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
- [13] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, London (1985)
- [14] Lowe, G.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
- [15] O'Halloran, C.M.: Acceptance based assurance. In: ASE 2001. IEEE (2001)
- [16] Ouaknine, J., Palikareva, H., Roscoe, A.W., Worrell, J.: Static Livelock Analysis in CSP. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 389–403. Springer, Heidelberg (2011)
- [17] Ouaknine, J.: Digitisation and Full Abstraction for Dense-Time Model Checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 37–51. Springer, Heidelberg (2002)
- [18] Ouaknine, J., Worrell, J.: Timed CSP = Closed Timed epsilon-automata. Nord. J. Comput. 10(2), 99–133 (2003)
- [19] Palikareva, H., Ouaknine, J., Roscoe, A.W.: SAT-solving in CSP trace refinement. Science of Computer Programming. Special issue on Automated Verification of Critical Systems (2011) (in press)
- [20] Reed, G., Roscoe, A.W.: A Timed Model for Communicating Sequential Processes. In: Kott, L. (ed.) ICALP 1986. LNCS, vol. 226, pp. 314–323. Springer, Heidelberg (1986)
- [21] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1998)
- [22] Roscoe, A.W.: Understanding Concurrent Systems. Springer (2011), <http://www.cs.ox.ac.uk/ucs/>
- [23] Roscoe, A.W., Huang, J.: Extending noninterference properties to the timed world. In: Proceedings of SAC 2006 (2006)

A Model Checker for Hierarchical Probabilistic Real-Time Systems*

Songzheng Song¹, Jun Sun², Yang Liu³, and Jin Song Dong⁴

- ¹ NUS Graduate School for Integrative Sci and Engineering, National University of Singapore
songsongzheng@nus.edu.sg
- ² Information System Technology and Design, Singapore University of Technology and Design
sunjun@sutd.edu.sg
- ³ Temasek Lab, National University of Singapore
tslliuya@nus.edu.sg
- ⁴ School of Computing, National University of Singapore
dongjs@comp.nus.edu.sg

Abstract. Real-life systems are usually hard to control, due to their complicated structures, quantitative time factors and even stochastic behaviors. In this work, we present a model checker to analyze hierarchical probabilistic real-time systems. A modeling language called PRTS is used to specify such systems, and automatic zone-abstraction approach, which is probability preserving, is used to generate finite state MDP. We have implemented PRTS in model checking framework PAT so that friendly user interface can be used to edit, simulate and verify PRTS models. Some experiments are conducted to show our tool's efficiency.

1 Introduction

Real-life systems could be complicated because of hierarchical structures and complex data operations; real-time behaviors are sometimes essential in such systems due to the interaction with the real world; in addition, unreliable environments could result in stochastic behaviors so that probability is necessary. These characteristics present the difficulty in properly designing and developing such systems. Applying model checking techniques in this domain is therefore very challenging, due to the requirements of an expressive enough modeling language as well as efficient model checking algorithms.

In this work, we present a new model checker to analyze complex systems. A modeling language Probabilistic Real-time System (PRTS) [16] is used to cover complicated system structures and data operations, real-time behavior and probability, meanwhile *dynamic zone abstraction* [16] is applied to handle the infinite state space caused by real-time factors. Different from zone abstraction used in other models such as Probabilistic Timed Automata (PTA) [10], our approach guarantees forward analysis after abstraction is precise. PRTS supports several widely used properties such as reachability checking, LTL checking and reward checking, with which users could analyze different aspects of the system. Our tool (public available at [1]) has been developed as

* This research was partially supported by research grant “SRG ISTD 2010 001” from Singapore University of Technology and Design and “MOE2009-T2-1-072” from MOE of Singapore.

a stand alone plug-in module in the verification framework PAT [14][1] to support the editing, simulation and verification of PRTS models with a friendly user interface.

Related Work. There are several model checkers exploring probabilistic real-time systems based on PTA. UPPAAL [4] supports real-time, concurrency and recently data operations as well as probability (UPPAAL-PRO), but lacks support for hierarchical control flow and is limited to maximal probabilistic reachability checking. PRISM [9] is popular in verifying systems having concurrency, probability and the combination of real-time and probability. However, it does not support hierarchical systems, but rather networks of flat finite state systems. Another tool mcpta [6] supports the verification of PTA by translating models into PRISM and only supports reachability checking. In addition, these tools only support simple data operations, which could be insufficient in modeling systems which have complicated structures and complex data operations.

2 Modeling with PRTS

In this section, we briefly introduce our modeling language PRTS, which extends Communicating Sequential Processes (CSP) with real-time and probabilistic behaviors.

Syntax. A subset of process constructors of PRTS are listed below to present its modeling abilities. Note that process constructors, like (conditional) choice, sequential and parallel compositions adopted from CSP for modeling hierarchical concurrent systems, are skipped due to the space limitation and readers can refer to [16] for details.

$$\begin{aligned} P = & a\{program\} \rightarrow P \mid Wait[d]P \text{ timeout}[d] Q \mid P \text{ interrupt}[d] Q \\ & \mid P \text{ deadline}[d] \mid P \text{ within}[d] \mid pcase\{pr_0 : P_0; pr_1 : P_1; \dots; pr_k : P_k\} \end{aligned}$$

Data Operation. PRTS supports shared memory models using global variables, which can be integer, boolean, integer array, and even arbitrary user-defined data structures. A user-defined data structure can be defined externally using programming languages like C#, Java, C and so on, and then imported into the model[1]. Data operations in PRTS are invoked through syntax $a\{program\} \rightarrow P$, which executes event a and $program$ simultaneously, and behaves as P afterwards.

Real-time. Several timed process constructors are supported in PRTS to capture the real-time behaviors of the system. Process $Wait[d]$ idles for d time units, where d is an integer constant. In $P \text{ timeout}[d] Q$, the first observable event of P shall occur before d time units elapse (since the process is *activated*). Otherwise, Q takes over control after d time units. $P \text{ interrupt}[d] Q$ behaves as P until d time units elapse, and then Q takes over control. PRTS extends Timed CSP [13] with additional timed process constructs. $P \text{ deadline}[d]$ constrains P to terminate before d time units. $P \text{ within}[d]$ requires that P must perform an observable event within d time units.

Probability. $pcase\{pr_0 : P_0; pr_1 : P_1; \dots; pr_k : P_k\}$ is used to model the randomized behaviors of a system. Here pr_i is a positive constant to express the probability weight. Intuitively, it means that with $\frac{pr_i}{pr_0+pr_1+\dots+pr_k}$ probability, the system behaves as P_i . Obviously the sum of all the probabilities in one $pcase$ is 1.

¹ Details can be found in PAT's user manual.

Note that the probabilistic real-time systems modeled in PRTS can be fully hierarchical, since P and Q in the above constructors can be any processes. This is different from PTA based languages which often have the form of a network of flat PTA.

Operational Semantics. The semantic model of PRTS is Markov Decision Processes (MDP) because of its mixture of non-deterministic and probabilistic choices. Note here we assume the valuations of variables and the processes reachable from the initial configuration are finite, therefore an MDP can have infinite states only due to its dense time transitions. In [16], we have defined *concrete firing rules* and *abstract firing rules* respectively. The former describes the operational semantics of PRTS, while the latter captures the execution behaviors of PRTS models after *zone abstraction*. This abstraction is necessary since it generates finite state space from a PRTS model so that traditional probabilistic model checking techniques can be used.

Our automatic zone abstraction approach is probability preserving for several properties such as reachability checking and LTL checking. A proof sketch is as follows: given a concrete MDP and one of its scheduler, a discrete-time Markov Chain (DTMC) can be defined; we can always build a corresponding DTMC in the abstract MDP to guarantee these two DTMC are time-abstract bi-similar, and vice versa [18][16].

We remark that forward reachability of PTA using zone abstraction is not accurate [10]. In PTA, given an abstract DTMC defined by the abstract model and a scheduler, it is possible that there is no corresponding concrete DTMC can be defined from the concrete model. Therefore the maximum (minimum) probability of reachability property in the abstract model is an upper (lower) bound of the accurate result. Some approaches such as [8] are used to solve this problem.

3 System Analysis

In our model checker, PRTS models can be analyzed by the built-in editor, simulator and verifier, through which we could investigate system behaviors of the models. In this section we briefly present how the simulator and verifier work.

3.1 Simulation

Our tool provides a discrete-event simulator which allows users to interactively and visually simulate system behaviors. In simulation, PRTS models follow the abstract operation semantics in order to guarantee that each step reflects a meaningful execution of the system. Users could choose automatic simulation, which means the simulator will randomly execute the model and generate the random states, or manual simulation, which allows users to choose next event from the current enabled events. Through simulation, users could visually check how the model executes step by step, which is very useful in system design and analysis, especially when there are some undesired executions found in verification. Simulation is a good complement to verification since users could have an intuitive observation and it makes debugging more convenient.

3.2 Verification

Compared with simulation, automatic verification plays a more important role in system analysis since it indicates the accurate result of whether a property is satisfied in a system. Two aspects are quite significant in verification with a model checker. One is the properties it can support, and the other is the efficiency of the verification algorithms. In the following, we review several widely used properties in PRTS, and some techniques in the verification algorithm to speed up the model checking procedure.

Properties Supported. PRTS supports multiple kinds of useful properties in system design since they are focusing on different aspects of the system. Because MDP has non-deterministic choices and (infinite) many schedulers, we consider the maximum and minimum probability of a specified property and mainly follow the algorithms in [3].

Reachability Checking. The maximum/minimum probability of reaching specific target states could be checked using numerical iterative method.

Reward Checking. The maximum/minimum accumulated rewards/costs to reach the target states could be calculated also through the iterative method. In PRTS we just consider the action reward, that is, assigning each visible action a reward which is a rational number.

LTL Checking. In PRTS we support LTL-X (LTL without ‘next’ operator) since in abstract model the semantics of ‘next’ is hard to define. In our setting, LTL formula can be built from not only atomic state propositions but also events so that it is called SE-LTL [5]. It is very expressive and suitable for PRTS since our language is both event-based and state-based. We adopt the Rabin automata-based approach to calculate the maximum/minimum probability that an SE-LTL is satisfied.

Refinement Checking. A desired property could be defined as a non-probability model and we can check a trace refinement relation between this model and the system specification [17].

Efficient Verification Techniques. In our implementation, after zone abstraction we adopt mainly two techniques to enhance the efficiency of verification.

Counter Abstraction. For some protocols having similar behaviors, we can group those processes together using counter abstraction [12][15]. Its extension to probabilistic system is still valid, whose proof is similar to work [7]. This approach reduces the state space without affecting the probability of specific properties which are irrelevant with processes identifiers.

Safety Checking via Refinement Checking. LTL formulas can be categorized into either safety or liveness [2]. In [17], we have proven that safety property can be verified via refinement checking. Given an SE-LTL property, our tool supports automatic safety detection. The experiment results show that sometimes it reduces verification time significantly compared with Rabin automata approach [17].

4 Implementation and Experiments

PRTS has been integrated into PAT, which is implemented with C# and can run on all widely-used operating systems. To make our tool more practical, we have developed a

Table 1. Experiments: Lift System

System	Random			Nearest		
	Result(pmax)	States	Time(s)	Result(pmax)	States	Time(s)
lift=2; floor=2; user=2	0.21875	20120	1.47	0.13889	12070	1.33
lift=2; floor=2; user=3	0.47656	173729	15.04	0.34722	83026	6.23
lift=2; floor=2; user=4	0.6792	777923	90.66	0.53781	308602	28.31
lift=2; floor=2; user=5	0.81372	2175271	406.29	0.68403	740997	85.29
lift=2; floor=3; user=2	0.2551	72458	5.13	0.18	38593	2.89
lift=2; floor=3; user=3	0.54009	1172800	150.20	0.427	500897	48.05
lift=2; floor=4; user=2	0.27	170808	13.06	0.19898	86442	6.11
lift=3; floor=2; user=2	0.22917	562309	86.88	0.10938	266621	34.25

Table 2. Compared with PRISM

System	Result	PAT		PRISM		
		States	Time(s)	States	Iterations	Time(s)
FA(10K)	0.94727	1352	0.15	1065	19	1.98
FA(20K)	0.99849	5030	0.13	8663	34	65.08
FA(30K)	0.99994	11023	0.45	34233	45	575.03
FA(300K)	>0.99999	726407	30.74	-	-	-
ZC(100)	0.49934	404	0.15	135	0	0.28
ZC(300)	0.01291	4813	0.65	2129	26	2.73
ZC(500)	0.00027	12840	2.39	10484	44	63.19
ZC(700)	1E-5	24058	5.78	31717	60	427.70

Visual Studio 2010 plug-in (available at [1]) to edit, simulate and verify PRTS models inside Visual Studio. Next, we demonstrate some experiments² to show the efficiency of our tool; the testbed is a PC running Windows XP with Intel P8700 CPU@2.53GHz and 2GB memory.

First, we use a multi-lift system to demonstrate the effectiveness of PRTS. Such system contains different components, e.g. lifts and buttons; it usually has timing requirements in service and users may have random behaviors. An interesting phenomena in such system is that a user presses the button outside the lifts, but one lift on the same direction passes by without serving him/her. This is possible since the lift which is assigned to serve this user is occupied by other users for a long time, and other lifts reach that user's floor first and pass by.

The experiments results are listed in Table 1. We analyze two kinds of task assignment mechanisms: assigning to nearest lift and assigning to a random lift. From the table we could conclude that the first mechanism is better, since it has a smaller probability to ignore users' requests and this is consistent with common sense.

Next, we compare our model checker with PRISM on verifying benchmark systems of probabilistic real-time system. Here we use two PTA models described in [8]. One

² Due to space constraint, detailed information of the models and properties can be found in [1].

is the *firewire abstraction* (*FA*) for IEEE 1394 FireWire root contention protocol and the other is *zeroconf* (*ZC*) for Zeroconf network configuration protocol. We build PTA models using PRISM and PRTS models using PAT, and verify the desired reachability properties to check the efficiency of these two tools. Here we choose PRISM’s default verification technique: stochastic games since it usually has the best performance [8].

The results are listed in Table 2. ‘-’ means that experiment takes more than 1 hour. The parameter of each model is the deadline constrain; for PRISM, *Iteration* means how many refinements the *stochastic game* approach executes to get the precise result. In these cases we notice PRTS is much faster than PRISM’s PTA since our approach just uses zone abstraction and theirs must have additional refinement procedure.

5 Conclusion

In this work, we proposed a model checker for hierarchical probabilistic real-time systems. Its effectiveness and efficiency are demonstrated through several case studies. As for future work, we are exploring more aspects of probabilistic real-time system such as zeno-check and digitization, and various properties such as real-time property.

References

1. PRTS Model Checker, <http://www.comp.nus.edu.sg/~pat/cav12ptrts>
2. Alpern, B., Schneider, F.B.: Recognizing Safety and Liveness. *Distributed Computing* 2(3), 117–126 (1987)
3. Baier, C., Katoen, J.: *Principles of Model Checking*. The MIT Press (2008)
4. Behrman, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST, pp. 125–126. IEEE (2006)
5. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
6. Hartmanns, A., Hermanns, H.: A modest approach to checking probabilistic timed automata. In: QEST, pp. 187–196 (September 2009)
7. 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)
8. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Games for Verification of Probabilistic Timed Automata. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 212–227. Springer, Heidelberg (2009)
9. 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)
10. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic Verification of Real-time Systems with Discrete Probability Distributions. *Theoretical Computer Science* 282(1), 101–150 (2002)
11. Liu, Y., Pang, J., Sun, J., Zhao, J.: Verification of population ring protocols in pat. In: TASE, pp. 81–89. IEEE Computer Society (2009)
12. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0, 1, \infty)$ -Counter Abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)

13. Schneider, S.: Concurrent and Real-time Systems. John Wiley and Sons (2000)
14. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
15. Sun, J., Liu, Y., Roychoudhury, A., Liu, S., Dong, J.S.: Fair Model Checking with Process Counter Abstraction. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 123–139. Springer, Heidelberg (2009)
16. Sun, J., Liu, Y., Song, S., Dong, J.S., Li, X.: PRTS: An Approach for Model Checking Probabilistic Real-Time Hierarchical Systems. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 147–162. Springer, Heidelberg (2011)
17. Sun, J., Song, S., Liu, Y.: Model Checking Hierarchical Probabilistic Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 388–403. Springer, Heidelberg (2010)
18. Sun, J., Song, S., Liu, Y., Dong, J.S.: PRTS: Specification and Model Checking. Technical report (2011), <http://www.comp.nus.edu.sg/~pat/preport.pdf>

SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs

Shuvendu K. Lahiri¹, Chris Hawblitzel¹,
Ming Kawaguchi², and Henrique Rebêlo³

¹ Microsoft Research, Redmond, WA, USA

² University of California, San Diego

³ Federal University of Pernambuco, Brazil

Abstract. In this paper, we describe SymDiff, a language-agnostic tool for equivalence checking and displaying semantic (behavioral) differences over imperative programs. The tool operates on an intermediate verification language Boogie, for which translations exist from various source languages such as C, C# and x86. We discuss the tool and the front-end interface to target various source languages. Finally, we provide a brief description of the front-end for C programs.

1 Introduction

An evolving software module undergoes several changes — refactoring, feature additions and bug fixes. Such changes may introduce compatibility bugs or regression bugs that are detected much later in the life-cycle of the software. There is a need for tools that can aid the developers understand the impact of changes faster. Such tools will complement “syntactic diff” tools such as WinDiff and GNU Diff on one hand, and regression testing based change validation (that provides limited coverage) on the other.

In this paper, we describe the design of SymDiff (**Sy**mbolic **D**iff), a semantic differencing tool for imperative programs. Unlike most existing equivalence checking tools for software, the tool operates on an intermediate verification language called Boogie [4] (and hence language-agnostic). This provides a separation of concerns — the core analysis algorithms are independent of source language artifacts (object-orientation, generics, pointer arithmetic etc.) and are therefore reusable across different languages. To perform scalable equivalence checking, we leverage the modular program verifier in Boogie that exploits the Satisfiability Modulo Theories (SMT) solver Z3 [4]. A novel feature of the tool is that it displays *abstract* counterexamples for equivalence proofs by highlighting intra-procedural traces in the two versions (see Figure 1), a semantic extension to differing source lines highlighted by syntactic diff tools.

Because of the language-agnostic nature of the tool, one only needs a translator from the source language (such as C) to Boogie (many of which already exist). We describe the front-end interface required from such translators to target SymDiff. Finally, we briefly describe the implementation of one such front-end for C programs.

```

7: #define ADD 1
8: #define SUB 2
9: #define MULT 3
10: #define DIV 4
11:
12: void Eval(PEXPR e)
13: {
14:     int op, a1, a2, res;
15:
16:     op = e->oper;
17:     [op = 3, e = 4413, e->oper = 3]
18:     a1 = e->op1;
19:     [a1 = 0, e = 4413, e->op1 = 0]
20:     a2 = e->op2;
21:     [a2 = -141, e = 4413, e->op2 = -141]
22:     res = -1;
23:     [res = -1]
24:
25:     switch (op)
26:     {
27:         case ADD:
28:             res = a1 + a2; break;
29:         case SUB:
30:             res = a1 - a2; break;
31:         case MULT:
32:             res = a1 * a2; break;
33:         default: break;
34:     }
35:     e->result = res;
36:     [e = 4413, e->result = 0, res = 0]
37:
38: }
39:
```

Fig. 1. Output of SymDiff for displaying semantic differences for C programs. The yellow source lines highlight a path, and the gray lines display values of some program expressions after each statement in the trace.

2 SymDiff

SymDiff operates on programs in an intermediate verification language Boogie [1]. Boogie is an imperative language consisting of assignments, assertions, control-flow and procedure calls. Variables (globals, locals and procedure parameters) and expressions can be either of a scalar type τ or a map type $[\tau']\tau$. We currently restrict SymDiff to the non-polymorphic subset of Boogie. A Boogie program may additionally contain symbolic constants, functions, and axioms over such constants and functions.

SymDiff takes as input two loop-free Boogie programs and a configuration file that *matches* procedures, globals, and constants from the two programs. Loops, if present, can be unrolled up to a user-specified depth, or may be extracted as tail-recursive procedures. The default configuration file matches procedures, parameters, returns and globals with the same name; the user can modify it to specify the appropriate configuration. The tool can (optionally) take a list of procedures that are assumed to be equivalent (e.g. procedures that do not call into any procedures with modifications). For each pair of matched procedures f_1 and f_2 , SymDiff checks for *partial equivalence* — terminating executions of f_1 and f_2 under the same input states result in identical output states. The input state of a procedure consists of the value of parameters and globals (hereafter referred to as a single variable gl) on entry, and the output state consists of the value of the globals and returns on exit.

```

procedure Eq.f1.f2(x){
    var g10;
    g10 := g1; //copy the globals
    r1 := inline call f1(x);
    g11 := g1; //store output globals
    g1 := g10; //restore globals
    r2 := inline call f2(x);
    g12 := g1; //store output globals
    assert (r1 == r2 && g11 == g12);
}

```

Fig. 2. Procedure for checking equivalence of f_1 and f_2

Given two programs P_1 and P_2 and a pairing of procedures over the two programs, the algorithm below checks for equivalence modularly. For each pair of paired procedures f_1 and f_2 , we create a new Boogie procedure $\text{Eq.}f_1.f_2$ (Figure 2) that checks partial equivalence of f_1 and f_2 . To enable modular checking, the procedure calls inside $\text{Eq.}f_1.f_2$ (i.e. callees of f_1 and f_2) are replaced by uninterpreted functions that update the modified globals and the return; the input to the functions are parameters and the globals that are read by the procedure. The resulting set of procedures $\{\text{Eq.}f_i.f_j \mid f_i \in P_1, f_j \in P_2\}$ (one for each matched pair f_i and f_j) are analyzed by the Boogie modular verifier using verification condition generation [1] and SMT solver Z3. We omit details of verification condition generation here; it suffices to know that it transforms a program (a set of annotated procedures) to a single logical formula whose validity implies that the program does not fail. In our case, if *all* the $\text{Eq.}f_i.f_j$ procedures are verified, then the matched procedure pairs in P_1 and P_2 are partially equivalent. On the other hand, if the assertion in any $\text{Eq.}f_i.f_j$ procedure cannot be proved by Boogie, we extract a set of intraprocedural paths through f_i and f_j and report them to the user. We modified Boogie to produce multiple (up to a user-specified limit) counterexample traces for the same assertion.

In addition to the purely modular approach, SymDiff offers various options for inlining callees (to improve precision at the cost of scalability) for the case of non-recursive programs. There are options for either inlining (a) every callee, (b) only the callees that can't be proved equivalent, or (c) only behaviors in callees that can't be proved equivalent (*differential inlining* [8]). These options require a bottom-up traversal of the call graph of procedures.

3 Interface for Source Languages

In this section, we briefly describe the important considerations for adapting SymDiff for a source imperative language such as C, C#, or x86. First, one needs a translator for the language (say C) that performs two tasks: (i) represents the state of a program (e.g. variables, pointers and the heap) explicitly in terms of scalar and map variables in Boogie, and (ii) translates each statement in the source language to a sequence of statements in Boogie. The precision and soundness of the resulting tool will be parameterized by how faithful the translator is. Many such translators exist today with various precision and soundness trade-offs. For example, HAVOC [3] translates C programs to Boogie; Spec# [2] converts C# programs to Boogie; there have also been translators from binary (x86) programs to Boogie [5].

When two procedures cannot be proven equivalent, SymDiff generates counterexample traces on the source programs (Figure 1). The counterexample contains an intra-procedural trace for each procedure and values of “relevant” program expressions (of scalar type) for each statement. We have found this to be the most useful feature of the tool when applied to real examples. This feature requires two pieces of (optional) additional information in the translated Boogie programs, for *each* source line translated:

- The source file and the line number have to be provided as attributes.
- For each scalar valued program expression e to be displayed in the trace (e.g. $e \rightarrow \text{oper}$ in Line 10 of the first program in Figure 1), associate the corresponding expression in Boogie.

Note that this requires only a *one time* change to the translator for the source language to Boogie.

Non-deterministic Statements. In the presence of non-deterministic statements (such as the Boogie statement `havoc x` that scrambles a variable x), a procedure may not be equivalent to itself. Source language translators often use non-deterministic statements such as `havoc` to model allocation, effect of I/O methods such as `scanf`, calls to external APIs etc. To use SymDiff effectively, we require that the translators use *deterministic* statements to model such cases. We provide an example of deterministic modeling of allocation for C programs in the next section. SymDiff also models external procedures as deterministic (in their parameters) transformers using uninterpreted functions.

3.1 C Front End

In this section, we briefly describe the implementation of the front-end for C programs. The tool takes two directories (for the two versions) containing a set of .c files and a makefile. We use the HAVOC [3] tool to translate C programs into Boogie programs. HAVOC uses maps to model the heap of the C program [3], where pointer and field dereferences are modeled as select or updates of a map. By default, HAVOC assumes that the input C programs are *field safe* (i.e. different field names cannot alias) and maintains a map per word-valued (scalar or pointer) field and type. For example, the statement `x->f := *(int*)y;` is modeled as `f[x+4] := T_int[y];` using two maps `f` and `T_int` of type `[int] int` (assuming offset of `f` is 4 inside `x`).

We modified HAVOC to incorporate deterministic modeling of allocation (for `malloc` and `free`) and I/O methods (such as `scanf`, `getc`) [8]. Here we sketch the modeling of allocation: we maintain a (ghost) global variable `allocvar`, which can be modified by calls to `malloc` and `free`. `malloc` is modeled as follows (in Boogie) : `malloc(n:int) returns (r:int) {r := allocvar; allocvar := newAlloc(allocvar, n);}`, where `newAlloc` is an uninterpreted function. The specification for `free` is similar. The modeling ensures that two identical sequences of `malloc` and `free` return the same (but arbitrary) sequence of pointers. This suffices for many examples, but can be incomplete in the presence

Table 1. Results on Siemens benchmarks. “Proc” stands for procedures, “Time” is the time taken by SymDiff to analyze all the “Changed” procedures.

Example	#LOC	#Proc	#Versions	# Changed procs (Avg)	Time (sec) (Avg)	# Paths (Avg)	Enum Time (sec) (Avg)
tcas	173	9	42	1.2	0.64	26.72	1.19
print_tokens	727	18	7	1.4	1.30	357.43	2.87
print_tokens2'	569	19	10	1.1	0.90	169.36	1.25
replace	563	21	32	1.1	0.96	20.38	2.11
schedule	412	18	9	1.1	0.94	16.00	1.99
schedule2	373	16	10	1	0.83	10.60	0.99
print_tokens2_n4	569	19	1	1	1.13	7560	56.83
print_tokens2_n6	569	19	1	1	1.30	>10,000	160.63
print_tokens2_e4	569	19	1	1	3.50	4200	24.02
print_tokens2_e6	569	19	1	1	32.96	>10,000	150.61

of different allocation orders. We also added an option to generate the information required to display the counterexample traces. For each statement, we generate any pointer or scalar subexpression in the trace. For example, for the C statement $x \rightarrow f.g \rightarrow h = y[i] + z.f$; we add the expressions $\{x, x \rightarrow f.g, x \rightarrow f.g \rightarrow h, y[i], z.f\}$ whose values will be displayed in the trace. For procedure calls, we add the expressions in the arguments and the return. Figure 1 shows the semantic diff as a pair of traces over two programs. The second program performs some *refactoring* and *feature addition* (case for MULT). A syntactic diff tool gets confused by the refactoring and offers little idea about the change in behavior.

Table 1 describes an evaluation of SymDiff on a set of medium-sized C programs representing the Siemens benchmarks from the SIR repository [10]. Each program comes with multiple versions representing injection of real and seeded faults. The benchmark `print_tokens2'` represents a slightly altered version of the `print_token2` benchmark, where we change a constant loop iterating 80 times to one over a symbolic constant n . The experiments were performed on a 3GHz Windows 7 machine with 16GB of memory. We used a loop unroll depth of 2 for the examples. The runtime of SymDiff (“Time”) does not include the time required to generate Boogie files. The number of intraprocedural paths (“Paths”) correspond to the number of feasible paths inside $E.f_1.f_2$ that reach the return statement, and “Enum Time” is the time inside Z3 to enumerate them using an ALL-SAT procedure. The first few rows indicate that the tool scales well for finding differences when the number of intraprocedural paths is less than 1000. To investigate the effect of large number of paths, we created two sets of examples `print_token2_n<k>` and `print_tokens2_e<k>` (with loop unrolling of k), for semantically different and equivalent procedures respectively. The results indicate that the tool scales better on semantically different procedures, perhaps due to the large number of paths leading to a difference. For the equivalent cases too, the scalability appears to be better than the approach of enumerating paths outside Z3. In addition to these examples, the tool has been successfully applied to C programs several thousand lines large.

4 Conclusion and Related Work

In this paper, we describe the design of a language-agnostic semantic differencing tool for imperative programs. We have currently developed a front-end for C programs. We have also built a preliminary front-end for x86 programs that we have applied to perform compiler validation. We are also developing a front-end for .NET programs using a variation of the Spec# tool chain. We are currently working on making a binary release of the tool along with the C front end at <http://research.microsoft.com/projects/syndiff/>.

Related Work. There have been a few recent static tools for performing semantic diff for programs. Jackson and Ladd [7] use dependencies between input and the output variables of a procedure — it does not use any theorem provers. The approach of regression verification [6] uses SMT solvers to check equivalence of C programs in the presence of mutual recursion, without requiring all procedures to be equivalent. This is the work closest to ours¹, and the main difference lies in the language agnostic nature of our tool, generation of abstract counterexamples, and the modeling of the heap. Differential symbolic execution [9] uses symbolic execution to enumerate paths to check for equivalence. Our preliminary experience shows that the use of verification conditions instead of path enumeration often helps SymDiff scale to procedures with several thousand intraprocedural paths.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., M. Leino, K.R., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: POPL, pp. 302–314 (2009)
4. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011)
6. Godlin, B., Strichman, O.: Regression verification. In: DAC, pp. 466–471 (2009)
7. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: ICSM, pp. 243–252 (1994)
8. Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research (2010)
9. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: SIGSOFT FSE, pp. 226–237 (2008)
10. Software-artifact Infrastructure Repository,
<http://sir.unl.edu/portal/index.html>

¹ Ofer Strichman has helped incorporate the algorithm into SymDiff.

Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems

Tool Paper

Sylvain Conchon¹, Amit Goel², Sava Krstić²,
Alain Mebsout¹, and Fatiha Zaïdi¹

¹ LRI, Université Paris Sud, CNRS, Orsay F-91405

² Strategic CAD Labs, Intel Corporation

Abstract. Cubicle is a new model checker for verifying safety properties of parameterized systems. It implements a parallel symbolic backward reachability procedure using Satisfiability Modulo Theories. Experiments done on classic and challenging mutual exclusion algorithms and cache coherence protocols show that Cubicle is effective and competitive with state-of-the-art model checkers.

1 Tool Overview

Cubicle is used to verify safety properties of *array-based systems*. This is a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes [10]. Cache coherence protocols and mutual exclusion algorithms are typical examples of such systems. Cubicle model-checks by a symbolic backward reachability analysis on infinite sets of states represented by specific simple formulas, called *cubes*.

Cubicle is an open source software based on theoretical work in [1] and [11]. It is inspired by and closely related to the model checker MCMT [12], from which, in addition to revealing the implementation details, it mainly differs in a more friendly input language and concurrent architecture.

Cubicle is written in OCaml. Its SMT solver is a tightly integrated, lightweight and enhanced version of Alt-Ergo [7]; and its parallel implementation relies on the Functory library [9]. Cubicle is available at <http://cubicle.lri.fr>.

2 System Description Language

Cubicle's input language is a typed version of Murφ [8] similar to the one of UCLID [6], rudimentary at the moment, but more user-friendly than MCMT and sufficiently expressive for typical parameterized systems.

A system is described in Cubicle by: (1) a set of type, variable, and array declarations; (2) a formula for the initial states; and (3) a set of transitions. It is parametrized by a set of *process identifiers*, denoted by the built-in type `proc`. Standard types `int`, `real`, and `bool` are also built in. Additionally, the user

can specify abstract types and enumerations with simple declarations like “`type data`” and “`type msg = Empty | Req | Ack`”. We show the language on the following `Mutex` example.

```

var Turn : proc
array Want[proc] : bool
array Crit[proc] : bool

init (z) {
    Want[z] = False && Crit[z] = False
}

unsafe (x y) {
    Crit[x] = True && Crit[y] = True
}

transition req (i)
requires { Want[i] = False }
{ Want[j] := case
    | i = j : True
    | _ : Want[j] }

```

```

transition enter (i)
requires { Want[i] = True
          && Crit[i] = False
          && Turn = i }
{ Crit[j] := case
    | i = j : True
    | _ : Crit[j] }

transition exit (i)
requires { Crit[i] = True }
{ Turn := . ;
  Crit[j] := case
    | i = j : False
    | _ : Crit[j] ;
  Want[j] := case
    | i = j : False
    | _ : Want[j] }

```

The system’s state is defined by a set of global variables and `proc`-indexed arrays. The initial states are defined by a universal conjunction of literals characterizing the values for some variables and array entries. A state of our example system `Mutex` consists of a process identifier `Turn` and two boolean arrays `Want` and `Crit`; a state is initial iff both arrays are constantly `false`.

Transitions are given in the usual guard/action form and may be parameterized by (one or more) process identifiers. They define the system’s execution: an infinite loop that at each iteration: (1) non-deterministically chooses a transition instance whose guard is true in the current state; and (2) updates state variables according to the action of the fired transition instance. Guards must be of the form $F \wedge \forall \bar{x}.(\Delta \Rightarrow F')$, where F, F' are conjunctions of literals (equations, disequations or inequations), and Δ says that every \bar{x} -variable is distinct from every parameter of the transition. Assignments can be non-deterministic, as in “`Turn := .`” in transition `exit` in `Mutex`. Array updates are coded by a case construct where each condition is a conjunction of literals, and the default case.

The safety property to be verified is expressed in its negated form as a formula that represents unsafe states. Each unsafe formula must be a *cube*, i.e., have the form $\exists \bar{x}.(\Delta \wedge F)$, where Δ is the conjunction of all disequations between the variables in \bar{x} , and F is a conjunction of literals. In the code, we leave the Δ part implicit. Thus in `Mutex`, the unsafe states are those in which `Crit[x]` and `Crit[y]` are true for two distinct process identifiers `x, y`.

3 Implementation Details and Optimizations

For a state formula Φ and a transition instance t , let $pre_t(\Phi)$ be the formula describing the set of states from which a Φ -state can be reached in one t -step.

Let also $\text{pre}(\Phi)$ be the union of $\text{pre}_t(\Phi)$ for all possible t . In its simplest form, the backward reachability algorithm constructs a sequence Φ_0, Φ_1, \dots such that Φ_0 is the system's unsafe condition and $\Phi_{i+1} = \Phi_i \vee \text{pre}(\Phi_i)$. The algorithm terminates with the first Φ_n that fails the *safety check* (consistency with the initial condition), or passes the *fixpoint check* $\Phi_n \vdash \Phi_{n-1}$.

In array-based systems, $\text{pre}_t(\phi)$ can be represented as a union (disjunction) of cubes, for every cube ϕ and every t . Thus, the Φ_i above are unions of cubes too, and the algorithm above can be modified to work only with cubes, as follows. Maintain a set V and a priority queue Q of *visited* and *unvisited cubes* respectively. Initially, let V be empty and let Q contain the system's unsafe condition. Then, at each iteration, take the highest-priority cube ϕ from Q and do the safety check for it, same as the above. If it fails, terminate with “system unsafe”. If the safety check passes, proceed to the *subsumption check* $\phi \vdash \bigvee_{\psi \in V} \psi$. If this fails, then add ϕ to V , compute all cubes in $\text{pre}_t(\phi)$ (for every t), add them to Q , and move on to the next iteration. If the subsumption check succeeds, then drop ϕ from consideration and move on. The algorithm terminates when a safety check fails or Q becomes empty. When an unsafe cube is found, Cubicle actually produces a counterexample trace.

Safety checks, being ground satisfiability queries, are easy for SMT solvers. The challenge is in subsumption checks $\phi \vdash \bigvee_{\psi \in V} \psi$ because of their size and the “existential implies existential” logical form. Assuming $\phi \triangleq \exists \bar{x}. F$ and $\psi \triangleq \exists \bar{y}. G_\psi$ ($\psi \in V$), the subsumption check translates into the validity check for the ground formula $H \triangleq (F \Rightarrow \bigvee_{\psi \in V} \bigvee_{\sigma \in \Sigma} (G_\psi)\sigma)$, where Σ is the set of all substitutions from \bar{y} to \bar{x} . Now, viewing any cube $G_\psi\sigma$ as a set of literals, one can make two useful comparisons with F : (1) if $G_\psi\sigma$ is a subset of F , then H is valid; (2) if $G_\psi\sigma$ contains a literal that directly contradicts a literal of F , then $G_\psi\sigma$ is redundant in H (can be removed without logically changing H). Cubicle aggressively attempts to prove H by building and verifying it incrementally, adding one disjunct to its consequent at a time. Essentially, it examines all pairs (ψ, σ) one-by-one, stopping the process when the current overapproximation of H becomes known to be valid. For each pair (ψ, σ) , the cube $G_\psi\sigma$ is first checked for redundancy; if redundant, it is ignored and a new pair (ψ, σ) is processed. If not redundant, the cube is subject to the subset check for $F \vdash G_\psi\sigma$. If this check succeeds, H is claimed valid; otherwise $G_\psi\sigma$ gets added to H (as a disjunct of its consequent) and the SMT solver checks if the newly obtained (weakened) H becomes valid.

Cubicle’s integration with the SMT solver at the API level is crucial for efficient treatment of the subsumption check. For any such check, a single context for the SMT solver is used; it just gets incremented and repeatedly verified. To support the efficient (symmetry-reduced) and exhaustive application of the inexpensive redundancy and subset checks, cubes are maintained in normal form where variables are renamed and implied literals removed at construction time.

The strategy for exploring the cube space is also essential. It pays to visit as few cubes as possible, which suggests giving priority to more “generic” cubes (those that represent larger sets of states). Thus, neither breadth-first nor depth-first search are good in their pure form. By default, Cubicle uses BFS (changeable

with the `-search` option to DFS or some variants) combined with a heuristically delayed treatment of some cubes. Currently, a cube is delayed if it introduces new process variables or does not contribute new information on arrays. Finally, Cubicle can remove cubes from \mathcal{V} when they become subsumed by a new cube.

Following MCMT, Cubicle supports user-supplied invariants and invariant synthesis, both of which can significantly reduce the search. Subsets of visited nodes that only contain predicates over a unique process variable are used as candidate invariants. Each of them is verified by starting a new resource limited backward reachability analysis. Cubicle can also discover “subtyping invariants” (saying that a variable can take only a selected subset of values) by a static analysis and these invariants can be natively exploited by the SMT solver which supports definitions of subtypes for enumerated data-types.

4 Multi-core Architecture

A natural way to scale up model checkers is to parallelize their CPU intensive tasks to take advantage of the widespread availability of multi-core machines or clusters [13][4][14]. In our framework, this is achieved by parallelizing the backward reachability loop and the generation of invariants. As mentioned above, since invariant synthesis is done independently from the main loop, it is straightforward to do it in parallel. However, concerning the loop itself, a naive parallel implementation would lose the precise guidance of the exploration¹, and more importantly, could break the correctness of the tool because of an unsafe use of some optimizations described in the previous section.

In our setting, the most resource consuming tasks are fixpoints checks which can be hard problems even for efficient SMT solvers. To gain efficiency, we implemented a concurrent version of BFS based on the observation that all such computations arising at the same level of the search tree can be executed in parallel. Our implementation is based on a centralized master/workers architecture. The master assigns fixpoints to workers and a synchronization barrier is placed at each level of the tree to retain a BFS order. The master asynchronously computes the preimages of nodes that are not verified as fixpoints by the workers. In the meanwhile, the master can also assign invariant generation tasks that will be processed by available workers. Finally, to safely delete nodes from \mathcal{V} , the master must discard the results about nodes that have been deleted while they were being checked by a worker.

Cubicle provides a concurrent breadth-first exploration of the search space using n parallel processes on a multi-core machine with the `-j n` option. The implementation is based on Functory [9], an OCaml library with a rich functional interface which facilitates the execution of parallel algorithms. Functory supports multi-core architectures and distributed networks; it has also a robust fault-tolerance mechanism. Concerning a distributed implementation, one of the main issues is to limit the size of data involved in transactions between the master and

¹ Our experiments showed that a non-deterministic parallel exploration can be worse than a guided sequential search.

the workers. For instance, the size of \mathcal{V} can quickly become a bottleneck in an architecture based on message passing communications. As future work, we plan to develop a distributed implementation that will only need to send updates of data-structures.

5 Experimental Results and Future Works

We have evaluated Cubicle on some classic and challenging mutual exclusion algorithms and cache coherence protocols. In the table below, we compare Cubicle's performances with state-of-the-art model checkers for parameterized systems. All benchmarks have been executed on a 64 bits machine with a quad-core Intel® Xeon® processor @ 3.2 GHz and 24 GB of memory. For each tool, we report the results obtained with the best settings we found. Note that the parallel version of Cubicle was run on 4 cores and that we only give its results for significantly time consuming problems. We denote by X benchmarks that we were unable to translate due to syntactic restrictions.

	Cubicle seq	Cubicle 4 cores	MCMT [12]	Undip [3]	PFS [2]
bakery	0.01s	-	0.01s	0.04s	0.01s
Dijkstra	0.24s	-	0.99s	0.04s	0.26s
Distributed_Lamport	2.3s	-	12.7s	unsafe	X
Java_Mlock	0.04s	-	0.06s	0.25s	0.02s
Ricart_Agrawala	1.8s	-	1m12s	4.3s	X
Szymanski_at	0.12s	-	0.71s	13.5s	timeout
Berkeley	0.01s	-	0.01s	0.01s	0.01s
flash_aggregated [15]	0.01s	-	0.02s	0.01s	X
German_Baukus	25.0s	17.1s	3h39m	9m43s	X
German_pfs	6m23s	3m8s	11m31s	timeout	47m22s
German_undip	0.17s	-	0.57s	1m32	X
Illinois	0.02s	-	0.04s	0.06s	0.06s
Moesi	0.01s	-	0.01s	0.01s	0.01s

Our experiments are very promising. They show first that the sequential version of Cubicle is competitive. The parallel version on 4 cores achieves speedups of 1.8 approximately, which is a good result considering the fact that cores cannot be fully exploited because of the synchronization required to perform a pertinent search. In practice, we found that the best setting for Cubicle is to use all the optimizations described in Section 3 (except for invariant synthesis which can be time consuming). In the table below, we show the respective effect of these optimizations on the version of the German protocol from [5] (German_baukus). In particular, it is worth noting that the subtyping analysis increases performances by an order of magnitude on this example.

delete nodes	Optimizations		Real Time (# nodes)	
	subtyping	invariant generation	sequential	4 cores
No	No	No	50m8s (22580)	27m13s (20710)
Yes	No	No	35m16s (20405)	19m39s (19685)
Yes	No	Yes	20m45s (15089)	13m55s (14527)
Yes	Yes	No	25.0s (3322)	17.1s (3188)

As future work, we would like to harness the full power of the SMT solver by sharing its data structures and even more tightly integrating its features in the model checker. In particular, this would be very useful to discover symmetries and to simplify nodes by finding semantic redundancies modulo theories. We are also interested in exploiting the unsat cores returned by the solver to improve our node deletion mechanism.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321 (1996)
2. Abdulla, P.A., Delzanno, G., Ben Henda, N., Rezine, A.: Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
3. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized Verification of Infinite-State Processes with Global Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
4. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: HiBi/PDMC, pp. 4–7 (2010)
5. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002)
6. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 78–92. Springer, Heidelberg (2002)
7. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantic combination of congruence closure with solvable theories. ENTCS 198(2), 51–69 (2008)
8. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: ICCD, pp. 522–525 (1992)
9. Filliâtre, J.-C., Kalyanasundaram, K.: Functory: A distributed computing library for Objective Caml. In: TFP, pp. 65–81 (2011)
10. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: IJCAR, pp. 67–82 (2008)
11. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. LMCS 6(4) (2010)
12. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: IJCAR, pp. 22–29 (2010)

13. Grumberg, O., Heyman, T., Ifergan, N., Schuster, A.: Achieving Speedups in Distributed Symbolic Reachability Analysis Through Asynchronous Computation. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 129–145. Springer, Heidelberg (2005)
14. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in Eddy. STTT 11(1), 13–25 (2009)
15. Park, S., Dill, D.L.: Protocol Verification by Aggregation of Distributed Transactions. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 300–310. Springer, Heidelberg (1996)

HybridSAL Relational Abstracter

Ashish Tiwari*

SRI International, Menlo Park, CA

ashish.tiwari@sri.com

Abstract. This paper describes the HybridSAL relational abstracter – a tool for verifying continuous and hybrid dynamical systems. The input to the tool is a model of a hybrid dynamical system and a safety property. The output of the tool is a discrete state transition system and a safety property. The correctness guarantee provided by the tool is that if the output property holds for the output discrete system, then the input property holds for the input hybrid system. The input is in HybridSal input language and the output is in SAL syntax. The SAL model can be verified using the SAL tool suite. This paper describes the HybridSAL relational abstracter – the algorithms it implements, its input, its strength and weaknesses, and its use for verification using the SAL infinite bounded model checker and k-induction prover.

1 Introduction

A dynamical system $(\mathbb{X}, \xrightarrow{a})$ with state space \mathbb{X} and transition relation $\xrightarrow{a} \subseteq \mathbb{X} \times \mathbb{X}$ is a *relational abstraction* of another dynamical system $(\mathbb{X}, \xrightarrow{c})$ if the two systems have the same state space and $\xrightarrow{c} \subseteq \xrightarrow{a}$. Since a relational abstraction contains all the behaviors of the concrete system, it can be used to perform safety verification.

HybridsAL relational abstracter is a tool that computes a relational abstraction of a hybrid system as described by Sankaranarayanan and Tiwari [8]. A hybrid system $(\mathbb{X}, \rightarrow)$ is a dynamical system with

- (a) state space $\mathbb{X} := \mathbb{Q} \times \mathbb{Y}$, where \mathbb{Q} is a finite set and $\mathbb{Y} := \mathbb{R}^n$ is the n -dimensional real space, and
- (b) transition relation $\rightarrow := \rightarrow_{cont} \cup \rightarrow_{disc}$, where \rightarrow_{disc} is defined in the usual way using guards and assignments, but \rightarrow_{cont} is defined by a system of *ordinary differential equation* and a *mode invariant*. One of the key steps in defining the (concrete) semantics of hybrid systems is relating a system of differential equation $\frac{dy}{dt} = f(\mathbf{y})$ with mode invariant $\phi(\mathbf{y})$ to a binary relation over \mathbb{R}^n , where \mathbf{y} is a n -dimensional vector of real-valued variables. Specifically, the semantics of such a system of differential equations is defined as:

$$\mathbf{y}_0 \rightarrow_{cont} \mathbf{y}_1 \text{ if there is a } t_1 \in \mathbb{R}^{\geq 0} \text{ and a function } F \text{ from } [0, t_1] \text{ to } \mathbb{R}^n \text{ s.t.}$$

* Supported in part by DARPA under subcontract No. VA-DSR 21806-S4 under prime contract No. FA8650-10-C-7075, and NSF grants CSR-0917398 and SHF:CSR-1017483.

$$\mathbf{y}_0 = F(0), \mathbf{y}_1 = F(t_1), \text{ and}$$

$$\forall t \in [0, t_1] : \left(\frac{dF(t)}{dt} = f(F(t)) \wedge \phi(F(t)) \right) \quad (1)$$

The concrete semantics is defined using the “solution” F of the system of differential equations. As a result, it is difficult to directly work with it.

The relational abstraction of a hybrid system $(\mathbb{X}, \xrightarrow{c}_{cont} \cup \xrightarrow{c}_{disc})$ is a discrete state transition system $(\mathbb{X}, \xrightarrow{a})$ such that $\xrightarrow{a} = \xrightarrow{a}_{cont} \cup \xrightarrow{c}_{disc}$, where $\xrightarrow{c}_{cont} \subseteq \xrightarrow{a}_{cont}$. In other words, the discrete transitions of the hybrid system are left untouched by the relational abstraction, and only the transitions defined by differential equations are abstracted.

The HybridSal relational abstracter tool computes such a relational abstraction for an input hybrid system. In this paper, we describe the tool, the core algorithm implemented in the tool, and we also provide some examples.

2 Relational Abstraction of Linear Systems

Given a system of linear ordinary differential equation, $\frac{d\mathbf{x}}{dt} = A\mathbf{x} + \mathbf{b}$, we describe the algorithm used to compute the abstract transition relation \xrightarrow{a} of the concrete transition relation \xrightarrow{c} defined by the differential equations.

The algorithm is described in Figure 1. The input is a pair (A, b) , where A is a $(n \times n)$ matrix of rational numbers and b is a $(n \times 1)$ vector of rational numbers. The pair represents a system of differential equations $\frac{d\mathbf{x}}{dt} = A\mathbf{x} + \mathbf{b}$. The output is a formula ϕ over the variables \mathbf{x}, \mathbf{x}' that represents the relational abstraction of $\frac{d\mathbf{x}}{dt} = A\mathbf{x} + \mathbf{b}$. The key idea in the algorithm is to use the eigenstructure of the matrix A to generate the relational abstraction.

The following proposition states the correctness of the algorithm.

Proposition 1. *Given (A, b) , let ϕ be the output of procedure `linODEabs` in Figure 1. If \rightarrow_{cont} is the binary relation defining the semantics of $\frac{d\mathbf{x}}{dt} = A\mathbf{x} + b$ with mode invariant True (as defined in Equation 1), then $\rightarrow_{cont} \subseteq \phi$.*

By applying the above abstraction procedure on the dynamics of each mode of a given hybrid system, the HybridSal relational abstracter constructs a relational abstraction of a hybrid system. This abstract system is a purely discrete infinite state space system that can be analyzed using infinite bounded model checking (inf-BMC), k-induction, or abstract interpretation.

We make two important remarks here. First, the relational abstraction constructed by procedure `linODEabs` is a Boolean combination of linear *and* nonlinear expressions. By default, HybridSal generates conservative linear approximations of these nonlinear relational invariants. HybridSal generates the (more precise) nonlinear abstraction (as described in Figure 1) when invoked using an appropriate command line flag. Note that most inf-BMC tools can only handle linear constraints. However, there is significant research effort going on into extending SMT solvers to handle nonlinear expressions. HybridSal relational abstracter and SAL inf-BMC have been used to create benchmarks for linear *and* nonlinear SMT solvers.

linODEabs(A, b): Input: a pair (A, b) , where $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^{n \times 1}$.

Output: a formula ϕ over the variables \mathbf{x}, \mathbf{x}'

1. identify all variables x_1, \dots, x_k s.t. $\frac{dx_i}{dt} = b_i$ where $b_i \in \mathbb{R} \quad \forall i$
let E be $\{\frac{x'_i - x_i}{b_i} \mid i = 1, \dots, k\}$
2. partition the variables \mathbf{x} into \mathbf{y} and \mathbf{z} s.t. $\frac{d\mathbf{x}}{dt} = A\mathbf{x} + \mathbf{b}$ can be rewritten as

$$\begin{bmatrix} \frac{d\mathbf{y}}{dt} \\ \frac{d\mathbf{z}}{dt} \end{bmatrix} = \begin{bmatrix} A_1 & A_2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix} + \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}$$

where $A_1 \in \mathbb{R}^{n_1 \times n_1}$, $A_2 \in \mathbb{R}^{n_1 \times n_2}$, $\mathbf{b}_1 \in \mathbb{R}^{n_1 \times 1}$, $\mathbf{b}_2 \in \mathbb{R}^{n_2 \times 1}$, and $n = n_1 + n_2$

3. set ϕ to be *True*
4. let \mathbf{c} be a real left eigenvector of matrix A_1 and let λ be the corresponding real eigenvalue, that is, $\mathbf{c}^T A_1 = \lambda \mathbf{c}^T$
5. if $\lambda == 0 \wedge c^T A_2 == 0$: set $E := E \cup \{\frac{c^T(\mathbf{y}' - \mathbf{y})}{c^T \mathbf{b}_1}\}$; else: $E := E$
6. if $\lambda \neq 0$: define vector \mathbf{d} and real number e as: $\mathbf{d}^T = \mathbf{c}^T A_2 / \lambda$ and $e = (\mathbf{c}^T \mathbf{b}_1 + \mathbf{d}^T \mathbf{b}_2) / \lambda$
let $p(\mathbf{x})$ denote the expression $\mathbf{c}^T \mathbf{y} + \mathbf{d}^T \mathbf{z} + e$ and let $p(\mathbf{x}')$ denote $\mathbf{c}^T \mathbf{y}' + \mathbf{d}^T \mathbf{z}' + e$
if $\lambda > 0$: set $\phi := \phi \wedge [(p(\mathbf{x}') \leq p(\mathbf{x}) < 0) \vee (p(\mathbf{x}') \geq p(\mathbf{x}) > 0) \vee (p(\mathbf{x}') = p(\mathbf{x}) = 0)]$
if $\lambda < 0$: set $\phi := \phi \wedge [(p(\mathbf{x}) \leq p(\mathbf{x}') < 0) \vee (p(\mathbf{x}) \geq p(\mathbf{x}') > 0) \vee (p(\mathbf{x}') = p(\mathbf{x}) = 0)]$
7. if there are more than one eigenvectors corresponding to the eigenvalue λ , then update ϕ or E by generalizing the above
8. repeat Steps (4)–(7) for each pair (\mathbf{c}, λ) of left eigenvalue and eigenvector of A_1
9. let $\mathbf{c} + \imath \mathbf{d}$ be a complex left eigenvector of A_1 corresponding to eigenvalue $\alpha + \imath \beta$
10. using simple linear equation solving as above, find $\mathbf{c}_1, \mathbf{d}_1, e_1$ and e_2 s.t. if p_1 denotes $\mathbf{c}^T \mathbf{y} + \mathbf{c}_1^T \mathbf{z} + e_1$ and if p_2 denotes $\mathbf{d}^T \mathbf{y} + \mathbf{c}_2^T \mathbf{z} + e_2$ then

$$\frac{d}{dt}(p_1) = \alpha p_1 - \beta p_2 \quad \frac{d}{dt}(p_2) = \beta p_1 + \alpha p_2$$

let p'_1 and p'_2 denote the primed versions of p_1, p_2

11. if $\alpha \leq 0$: set $\phi := \phi \wedge (p_1^2 + p_2^2 \geq p'_1^2 + p'_2^2)$
if $\alpha \geq 0$: set $\phi := \phi \wedge (p_1^2 + p_2^2 \leq p'_1^2 + p'_2^2)$
12. repeat Steps (9)–(11) for every complex eigenvalue eigenvector pair
13. set $\phi := \phi \wedge \bigwedge_{e_1, e_2 \in E} e_1 = e_2$; return ϕ

Fig. 1. Algorithm implemented in HybridSal relational abstracter for computing relational abstractions of linear ordinary differential equations

Second, Procedure **linODEabs** can be extended to generate even more precise *nonlinear* relational abstractions of linear systems. Let p_1, p_2, \dots, p_k be k (linear and nonlinear) expressions found by Procedure **linODEabs** that satisfy the equation $\frac{dp_i}{dt} = \lambda_i p_i$. Suppose further that there is some λ_0 s.t. for each i $\lambda_i = n_i \lambda_0$ for some integer n_i . Then, we can extend ϕ by adding the following relation to it:

$$p_i(\mathbf{x}')^{n_j} p_j(\mathbf{x})^{n_i} = p_j(\mathbf{x}')^{n_i} p_i(\mathbf{x})^{n_j} \tag{2}$$

However, since p_i 's are linear or quadratic expressions, the above relations will be highly nonlinear unless n_i 's are small. So, they are not currently generated

by the relational abstracter. It is left for future work to see if good and useful linear approximations of these highly nonlinear relations can be obtained.

3 The HybridSal Relational Abstracter

The HybridSal relational abstracter tool, including the sources, documentation and examples, is freely available for download [10].

The input to the tool is a file containing a specification of a hybrid system and safety properties. The HybridSal language naturally extends the SAL language by providing syntax for specifying ordinary differential equations. SAL is a guarded command language for specifying discrete state transition systems and supports modular specifications using synchronous and asynchronous composition operators. The reader is referred to [7] for details. HybridSal inherits all the language features of SAL. Additionally, HybridSal allows differential equations to appear in the model as follows: if x is a real-valued variable, a differential equation $\frac{dx}{dt} = e$ can be written by assigning e to the dummy identifier $x\dot{}$. Assuming two variables x, y , the syntax is as follows:

```
guard(x,y) AND guard2(x,x',y,y') --> xdot' = e1; ydot' = e2
```

This represents the system of differential equations $\frac{dx}{dt} = e1, \frac{dy}{dt} = e2$ with mode invariant $guard(x, y)$. The semantics of this guarded transition is the binary relation defined in Equation II conjuncted with the binary relation $guard2(x, x', y, y')$. The semantics of all other constructs in HybridSal match exactly the semantics of their counterparts in SAL.

Figure 2 contains sketches of two examples of hybrid systems modeled in HybridSal. The example in Figure 2(left) defines a module **SimpleHS** with two real-valued variables x, y . Its dynamics are defined by $\frac{dx}{dt} = -y + x, \frac{dy}{dt} = -y - x$ with mode invariant $y \geq 0$, and by a discrete transition with guard $y \leq 0$. The HybridSal file **SimpleEx.hsal** also defines two safety properties. The latter one says that x is always non-negative. This model is analyzed by abstracting it

```
bin/hsal2hasal examples/SimpleEx.hsal
```

to create a relational abstraction in a SAL file named **examples/SimpleEx.sal**, and then (bounded) model checking the SAL file

```
sal-inf-bmc -i -d 1 SimpleEx helper
sal-inf-bmc -i -d 1 -l helper SimpleEx correct
```

The above commands prove the safety property using k -induction: first we prove a lemma, named **helper**, using 1-induction and then use the lemma to prove the main theorem named **correct**.

The example in Figure 2(right) shows the sketch of a model of the train-gate-controller example in HybridSal. All continuous dynamics are moved into one module (named **timeElapse**). The **train**, **gate** and **controller** modules define the state machines and are pure SAL modules. The **observer** module is also a pure SAL module and its job is to enforce synchronization between modules on events. The final system is a complex composition of the base modules.

The above two examples, as well as, several other simple examples are provided in the HybridSal distribution to help users understand the syntax and working

```

SimpleEx: CONTEXT = BEGIN
SimpleHS: MODULE = BEGIN
  LOCAL x,y: REAL
  INITIALIZATION
    x = 1; y IN {z:REAL| z <= 2}
  TRANSITION
    [ y >= 0 AND y' >= 0 -->
      xdot' = -y + x ;
      ydot' = -y - x
    [] y <= 0 --> x' = 1; y' = 2]
  END;
  helper: LEMMA SimpleHS |- G(0.9239*x >= 0.3827*y);
  correct : THEOREM
    SimpleHS |- G(x >= 0);
END

TGC: CONTEXT = BEGIN
Mode: TYPE = {s1, s2, s3, s4};
timeElapse: MODULE = BEGIN
  variable declarations
  INITIALIZATION x = 0; y = 0; z = 0
  TRANSITION
    [mode invariants -->
      --> xdot' = 1; ydot' = 1; zdot' = 1]
  END;
train: MODULE = ...
gate: MODULE = ...
controller: MODULE = ...
observer: MODULE = ...
system: MODULE = (observer || (train []
  gate [] controller [] timeElapse));
correct: THEOREM system |- G ( ... ) ;
END

```

Fig. 2. Modeling hybrid systems in HybridSal: A few examples

of the relational abstracter. A notable (nontrivial) example in the distribution is a hybrid model of an automobile's automatic transmission from [2]. Users have to separately download and install SAL model checkers if they wish to analyze the output SAL files using k-induction or infinite BMC.

The HybridSal relational abstracter constructs abstractions compositionally; i.e., it works on each mode (each system of differential equations) separately. It just performs some simple linear algebraic manipulations and is therefore very fast. The bottleneck step in our tool chain is the inf-BMC and k-induction step, which is orders of magnitude slower than the abstraction step (Table II).

4 Related Work and Conclusion

The HybridSal relational abstracter is a tool for verifying hybrid systems. The other common tools for hybrid system verification consist of (a) tools that iteratively compute an overapproximation of the reachable states [5], (b) tools that directly search for correctness certificates (such as inductive invariants or Lyapunov function) [9], or (c) tools that compute an abstraction and then analyze the abstraction [6][13]. Our relational abstraction tool falls in category (c), but unlike all other abstraction tools, it does not abstract the state space, but abstracts only the transition relation. In [8] we had defined relational abstractions and proposed many different techniques (not all completely automated at that time) to construct the relational abstraction.

The key benefit of relational abstraction is that it cleanly separates reasoning on continuous dynamics (where we use control theory or systems theory) and reasoning on discrete state transition systems (where we use formal methods.) The former is used for constructing high quality relational abstractions and the latter is used for verifying the abstract system.

Table 1. Performance on the 27 navigation benchmarks [4]: The HybridSal models, on purpose, enumerate all modes explicitly so that it becomes clear that the time (RA) for constructing relational abstraction grows linearly with the number of modes (modes). Inf-bmc starts to time out (TO) at 5 minutes at depth (d) 20 for examples with ≥ 25 modes. Ideally, one wants to perform inf-bmc with depth equal to number of modes. N100 means inf-bmc returned after 100 seconds with no counter-examples and C160 means inf-bmc returned after 160 seconds with a counter-example.

nav	1-5	6	7-8	9	10-11	12	13-15	16-18	19-21	22-24	25-27
modes	9	9	16	16	25	25	42	81	144	225	400
RA	2	2	3	3	5	5	9	20	40	80	180
d=4	N0	N0	N1	N1	N1	C1	N1	N2	N4	N6	N20
d=8	N1	C2	C100	C5	C10	C15	N20	N10	N25	N10	N60
d=12	N5	C3	TO	C18	C20	C50	C150	N10	TO	N40	T0
d=16	N40	C10	TO	C50	C50	C180	TO*	240*	TO	TO	TO
d=20	N100	C80	TO	C160	C80	TO	TO	TO	TO	TO	TO

We note that our tool is the first relational abstracter for hybrid systems and is under active development. We hope to enhance the tool by improving precision of the abstraction using mode invariants and other techniques, providing alternative to inf-bmc, and handling nonlinear differential equations.

References

1. Alur, R., Dang, T., Ivančić, F.: Counter-Example Guided Predicate Abstraction of Hybrid Systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 208–223. Springer, Heidelberg (2003)
2. Chutinan, A., Butts, K.R.: SmartVehicle baseline report: Dynamic analysis of hybrid system models for design validation. Ford Motor Co., Tech. report, Open Experimental Platform for DARPA MoBIES, Contract F33615-00-C-1698 (2002)
3. Clarke, E., Fehnker, A., Han, Z., Krogh, B.H., Stursberg, O., Theobald, M.: Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 192–207. Springer, Heidelberg (2003)
4. Fehnker, A., Ivančić, F.: Benchmarks for Hybrid Systems Verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004)
5. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
6. Hybridsal: Modeling and abstracting hybrid systems,
<http://www.cs1.sri.com/users/tiwari/HybridSalDoc.ps>
7. The SAL intermediate language, Computer Science Laboratory, SRI International, Menlo Park, CA (2003), <http://sal.cs1.sri.com/>

8. Sankaranarayanan, S., Tiwari, A.: Relational Abstractions for Continuous and Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 686–702. Springer, Heidelberg (2011)
9. Sturm, T., Tiwari, A.: Verification and synthesis using real quantifier elimination. In: ISSAC 2011, pp. 329–336 (2011)
10. Tiwari, A.: Hybridsal relational abstracter,
<http://www.cs1.sri.com/~tiwari/relational-abstraction/>

EULER: A System for Numerical Optimization of Programs*

Swarat Chaudhuri¹ and Armando Solar-Lezama²

¹ Rice University

² MIT

Abstract. We give a tutorial introduction to EULER, a system for solving difficult optimization problems involving programs.

1 Introduction

This paper is a tutorial introduction to EULER, a system for solving unconstrained optimization problems of the following form:

Let P be a function that is written in a standard C-like language, and freely uses control constructs like loops and conditional branches. Find an input \mathbf{x} to P such that the output $P(\mathbf{x})$ of P is *minimized* with respect to an appropriate distance measure on the space of outputs of P .

Many problems in software engineering are naturally framed as optimization questions like the above. While it may appear at first glance that a standard optimization package could solve such problems, this is often not so. For one, “white-box” optimization approaches like linear programming are ruled out here because the objective functions that they permit are too restricted. As for “black-box” numerical techniques like gradient descent or simplex search, they are applicable in principle, but often not in practice. The reason is that these methods work well only in relatively smooth search spaces; in contrast, branches and loops can cause even simple programs to have highly irregular, ill-conditioned behavior [1] (see Sec. 4 for an example). These challenges are arguably why numerical optimization has found so few uses in the world of program engineering.

The central thesis of the line of work leading to EULER is that *program approximation* techniques from program analysis can work together with blackbox optimization toolkits, and make it possible to solve problems where programs are the targets of optimization. Thus, programs do not have to be black boxes, but neither do they have to fall into the constricted space of what is normally taken to be white-box optimization. Specifically, the algorithmic core of EULER is *smooth interpretation* [12], a scheme for approximating a program by a series of smooth mathematical functions. Rather than the original program, it is these smooth approximations that are used as the targets of numerical optimization. As we show in Sec. 4, the result is often vastly improved quality of results.

* This work was supported by NSF Awards #1156059 and #1116362.

```

double parallel () {
    Error = 0.0;
    for(t = 0; t < T; t += dT) {
        if(stage==STRAIGHT) {           // <-- Drive in reverse
            if(t > ??) stage= INTURN; }   // Parameter  $t_1$ 
        if(stage==INTURN) {             // <-- Turn the wheels towards the curb
            car.ang = car.ang - ??;      // Parameter  $A_1$ 
            if(t > ??) stage= OUTTURN; } // Parameter  $t_2$ 
        if(stage==OUTTURN) {           // <-- Turn the wheels away from the curb
            car.ang = car.ang + ??;      // Parameter  $A_2$ 
            simulate_car(car); }
        Error = check_destination(car); // <-- Compute the error as the difference
                                         // between the desired and actual
                                         // positions of the car
    }
    return Error ;
}

```

Fig. 1. Sketch of a parallel parking controller

2 Using EULER

Programming EULER: Parallel Parking. EULER can be downloaded from <http://www.cs.rice.edu/~swarat/Euler>. Now we show to use the system to solve a program synthesis problem that reduces to numerical optimization.

The goal here is to design a controller for parallel-parking a car. The programmer knows what such a controller should do at a high level: it should start by driving the car in reverse, then at time t_1 , it should start turning the wheels towards the curb (let us say at an angular velocity A_1) and keep moving in reverse. At a subsequent time t_2 , it should start turning the wheels away from the curb (at velocity A_2) until the car reaches the intended position. However, the programmer does not know the optimal values of t_1 , t_2 , A_1 , and A_2 , and the system must synthesize these values. More precisely, let us define an objective function that determines the quality of a parallel parking attempt in terms of the error between the final position of the car and its intended position. The system’s goal is to minimize this function.

To solve this problem using EULER, we write a parameterized program—a *sketch*—that reflects the programmer’s partial knowledge of parallel parking.¹ The core of this program is the function `parallel` shown in Fig. 1. For space reasons, we omit the rest of the program—however, the complete sketch is part of the EULER distribution.

It is easy to see that `parallel` encodes our partial knowledge of parallel parking. The terms `??` in the code are “holes” that correspond, in order, to the parameters t_1 , A_1 , t_2 , and A_2 , and EULER will find appropriate values for them. Note that we can view `parallel` as a function `parallel(t_1, A_1, t_2, A_2)`. This is the function that EULER is to minimize.

¹ The input language of EULER is essentially the same as in the SKETCH programming synthesis system [4].

Occasionally, a programmer may have some insights about the range of optimal values for a program parameter. These may be communicated using holes of the form `??(p,q)`, where (p,q) is a real interval. If such an annotation is offered, EULER will begin its search for the parameter from the region (p, q) . Note, however, that this is no more than a “hint” to guide the search. EULER performs unconstrained optimization, and it is not guaranteed that the value finally found for the parameter will lie in this region. We leave the task of extending EULER to constrained optimization for future work.

Running EULER. Suppose we have built EULER and set up the appropriate library paths (specifically, EULER requires GSL—the GNU Scientific Library), and that our sketch of the parallel parking controller has been saved in a file `parallelPark.sk`. To perform our optimization task, we compile the file by issuing the command `"$ euler parallelPark.sk"`. This produces an executable `parallelPark.out`. Upon running this executable (`"$./parallelPark.out"`), we obtain an output of the following form:

```
Parameter #1: -37.0916;          Parameter #2: 19.4048;
Parameter #3: -41.1728;          Parameter #4: 1.11344;
Optimal function value: 10.6003
```

That is, the optimal value for t_1 is -37.0916, that for A_1 is 19.4048, and so on.

As mentioned earlier, EULER uses a combination of smooth interpretation and a blackbox optimization method. In the present version of EULER, the latter is fixed to be the Nelder-Mead simplex method [3], a derivative-free nonlinear optimization technique. However, we also allow the programmer to run, without the support of smoothing, every optimization method available in GSL. For example, to run the Nelder-Mead method without smoothing, the user issues the command `"$./parallelPark.out -nosmooth -method neldermead"`. For other command-line flags supported by the tool, run `"$./parallelPark.out -help"`.

3 System Internals

Now we briefly examine the internals of EULER. First, we recall the core ideas of smooth interpretation [12].

The central idea of smooth interpretation is to transform a program via *Gaussian smoothing*, a signal processing technique for attenuating noise and discontinuities in real-world signals. A blackbox optimization method is now applied to this “smoothed” program. Consider a program whose denotational semantics is a function $P : \mathbb{R}^k \rightarrow \mathbb{R}$: on input $\mathbf{x} \in \mathbb{R}^k$, the program terminates and produces the output $P(\mathbf{x})$. Also, let us consider Gaussian functions $\mathcal{N}_{\mathbf{x}, \beta}$ with mean $\mathbf{x} \in \mathbb{R}^k$ and a fixed standard deviation $\beta > 0$. Smooth interpretation aims to compute a function \overline{P} equaling the convolution of P_β and $\mathcal{N}_{\mathbf{x}, \beta}$: $\overline{P}_\beta(\mathbf{x}) = \int_{\mathbf{y} \in \mathbb{R}^k} P(\mathbf{y}) \mathcal{N}_{\mathbf{x}, \beta}(\mathbf{y}) d\mathbf{y}$.

For example, consider the program `“z := 0; if (x1 > 0 ∧ x2 > 0) then z := z – 2”` where \mathbf{z} is the output and \mathbf{x}_1 and \mathbf{x}_2 are inputs. The (discontinuous)

semantic function of the program is graphed in Fig. 2-(a). Applying Gaussian convolution to this function gives us a smooth function as in Fig. 2-(b).

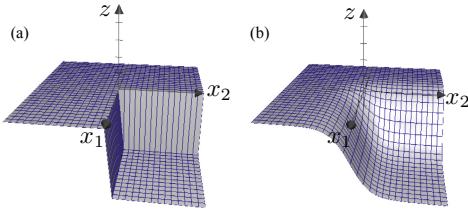


Fig. 2. (a) A discontinuous program (b) Gaussian smoothing

As computing the exact convolution of an arbitrary program is undecidable, any algorithm for program smoothing must introduce additional approximations. In prior work, we gave such an algorithm [1]—this is what EULER implements.

We skip the details of this algorithm here. However, it is worth noting that function \bar{P}_β is parameterized by the standard deviation

β of $\mathcal{N}_{x,\beta}$. Intuitively, β controls the extent of smoothing: higher values of β lead to greater smoothing and easier numerical search, and lower values imply closer correspondence between P and \bar{P}_β , and therefore, greater accuracy of results. Finding a “good” value of β thus involves a tradeoff. EULER negotiates this tradeoff by starting with a moderately high value of β , optimizing the resultant smooth function, then iteratively reducing β and refining the search results.

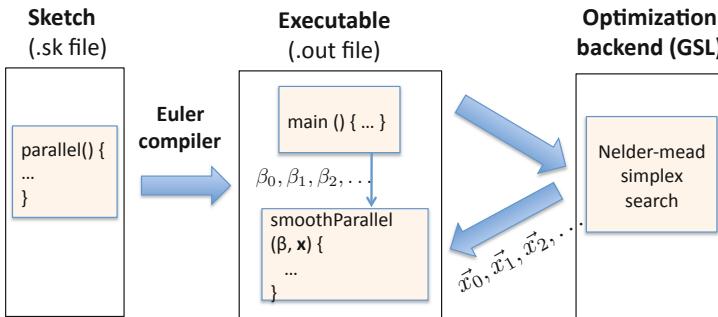


Fig. 3. Control flow in EULER

The high-level control flow in the tool is as in Fig. 3. Given an input file (say `parallelPark.sk`), the EULER compiler produces an executable called `parallelPark.out`. In the latter, the function `parallel(x)` has been replaced by a smooth function `smoothParallel(β, x)`. When we execute `parallelPark.out`, β is set to an initial value β_0 , and the optimization backend (GSL) is invoked on the function `smoothParallel(β = β₀, x)`. The optimization method repeatedly queries `smoothParallel`, starting with a random initial input x_0 and perturbing it iteratively in subsequent queries, and finally returns a minimum to the top-level loop. At this point, β is set to a new value and the same process is repeated. We continue the outer loop until it converges or there is a timeout.

4 Results

Now we present the results obtained by running EULER on our parallel parking example. These results are compared with the results of running the Nelder-Mead method (NM), without smoothing, on the problem.

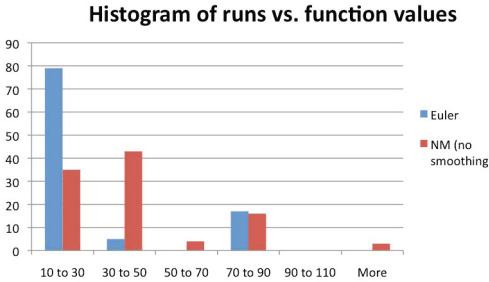


Fig. 4. Percentages of runs that lead to specific ranges of $\text{parallel}(\mathbf{x}_{\min})$ (lower ranges are better)

values of $\text{parallel}(\mathbf{x}_{\min})$ on all these runs have been clustered into several intervals, and the number of runs leading to outputs in each interval plotted as a histogram.

As we see, a much larger percentage of runs in EULER lead to lower (i.e., better) values of $\text{parallel}(\mathbf{x}_{\min})$. The difference appears starker when we consider the number of runs that led to the best-case behavior for the two methods. The best output value computed by both methods was 10.6003; however, 68 of the 100 runs of EULER resulted in this value, whereas NM had 26 runs within the range 10.0-15.0.

Let us now see what these different values of $\text{parallel}(\mathbf{x}_{\min})$ actually mean. We show in Fig. 6-(a) the trajectory of a car on an input \mathbf{x} for which the value $\text{parallel}(\mathbf{x})$ equals 40.0 (the most frequent output value, rounded to the first decimal, identified by NM). The initial and final positions of the car are marked; the two black rectangles represent other cars; the arrows indicate the directions that the car faces at different points in its trajectory. Clearly, this parking job would earn any driving student a failing grade.

On the other hand, Fig. 6-(b) shows the trajectory of a car parked using EULER on an input for which the output is 10.6, the most frequent output value found by EULER. Clearly, this is an excellent parking job.

As any local optimization algorithm starts its search from a random point \mathbf{x}_0 , the minima \mathbf{x}_{\min} computed by EULER and NM are random as well. However, the difference between the two approaches becomes apparent when we consider the *distribution* of $\text{parallel}(\mathbf{x}_{\min})$ in the two methods. In Fig. 4, we show the results of EULER and NM on 100 runs from initial points generated by a uniform random sampling of the region $-10 < t_1, A_1, t_2, A_2 < 10$. The

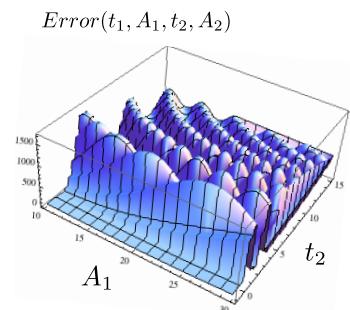


Fig. 5. Landscape of numerical search for parallel parking algorithm

The reason why NM fails becomes apparent when we examine the search space that it must navigate here. In Fig. 5, we plot the function `parallel(x)` for different values of t_2 and A_1 (t_1 and A_2 are fixed at optimal values). Note that the search space is rife with numerous discontinuities, plateaus, and local minima. In such extremely irregular search spaces, numerical methods are known not to work—smoothing works by making the space more regular. Unsurprisingly, similar phenomena were observed when we compared EULER with other optimization techniques implemented in GSL.

These observations are not specific to parallel parking—similar effects are seen on other parameter synthesis benchmarks [1] that involve controllers with discontinuous switching. Several of these benchmarks—including a model of a thermostat and a model of a gear shift—are part of the EULER distribution. More generally, the reason why `parallel` is so ill-behaved is fundamental: even simple programs may contain discontinuous if-then-else statements, which can be piled on top of each other through composition and loops, causing exponentially many discontinuous regions. Smooth interpretation is only the first attempt from the software community to overcome these challenges; more approaches to the problem will surely emerge. Meanwhile, readers are welcome to try out EULER and advise us on how to improve it.

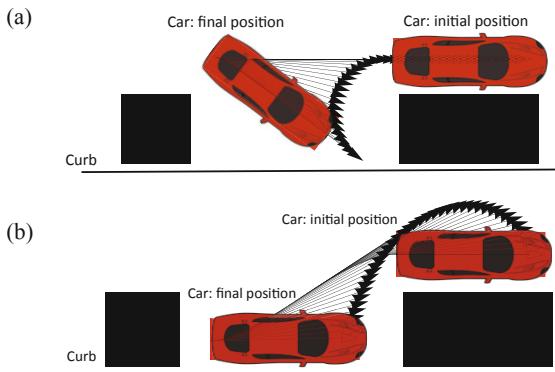


Fig. 6. (a) Parallel parking as done by NM; (b) Parallel parking as done by EULER

References

1. Chaudhuri, S., Solar-Lezama, A.: Smooth interpretation. In: PLDI, pp. 279–291 (2010)
2. Chaudhuri, S., Solar-Lezama, A.: Smoothing a Program Soundly and Robustly. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 277–292. Springer, Heidelberg (2011)
3. Nelder, J.A., Mead, R.: A simplex method for function minimization. The Computer Journal 7(4), 308 (1965)
4. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294. ACM (2005)

SPT: Storyboard Programming Tool^{*}

Rishabh Singh and Armando Solar-Lezama

MIT CSAIL, Cambridge, MA, USA

Abstract. We present SPT, a tool that helps programmers write low-level data-structure manipulations by combining various forms of insights such as abstract and concrete input-output examples as well as implementation skeletons. When programmers write such manipulations, they typically have a clear high-level intuition about how the manipulation should work, but implementing efficient low-level pointer manipulating code is error-prone. Our tool aims to bridge the gap between the intuition and the corresponding implementation by automatically synthesizing the implementation. The tool frames the synthesis problem as a generalization of an abstract-interpretation based shape analysis, and represents the problem as a set of constraints which are solved efficiently by the SKETCH solver. We report the successful evaluation of our tool on synthesizing several linked list and binary search tree manipulations.

1 Introduction

When programmers write data-structure manipulations, they typically have clear high-level visual insights about how the manipulation should work, but the translation of these insights to efficient low-level pointer manipulating code is difficult and error prone. Program synthesis [1][5][6] offers an opportunity to improve productivity by automating this translation. This paper describes our tool SPT¹ (Storyboard Programming Tool) that helps programmers write low-level implementations of data-structure manipulations by combining various forms of insights, including abstract and concrete input-output examples as well as implementation skeletons.

Our tool is based on a new synthesis algorithm [4] that combines abstract-interpretation based shape-analysis [2][3] with constraint-based synthesis [5][7][8]. The algorithm uses an abstraction refinement based approach to concisely encode synthesis constraints obtained from shape analysis.

In this paper, we present a high-level storyboard language that allows programmers to succinctly express the different elements that make up a storyboard. The language is more concise than the one described in [4] thanks to the use of inference to derive many low-level details of the storyboard. The paper also describes the architecture of the Storyboard Programming Tool and presents some new results comparing SPT with the Sketch synthesis system.

^{*} Supported by NSF under grant CCF-1116362.

¹ The Storyboard tool and benchmarks are available for download at
<http://people.csail.mit.edu/rishabh/storyboard-website/>

2 Overview: Linked List Deletion

We present an overview of our tool using linked list deletion as a running example. The goal of this manipulation is to delete a node pointed to by a variable `y` from an acyclic singly linked list. The manipulation iterates over the list until it finds the required node and then performs a sequence of pointer assignments to delete the node. SPT synthesizes an imperative implementation of deletion from a high-level storyboard in about two minutes.

The storyboard that SPT takes as input is composed of *scenarios*, *inductive definitions* and a *loop skeleton*. A scenario describes the (potentially abstract) state of a data-structure at different stages of the manipulation. Each scenario contains at least two configurations: `input` and `output` corresponding to the state of the data-structure before and after the manipulation; a scenario may also contain descriptions of the state at intermediate points in the computation.

The scenarios for linked list deletion are shown in Figure 11(a), and the corresponding visual description of the scenarios is shown in Figure 11(b). The first scenario `s1` describes an abstract input-output example, where the input list consists of two *summary nodes* `front` and `back` and a concrete node `ly` that is to be deleted. In this case, the summary node serves as an abstract representation of a list of arbitrary size. In general, a summary node represents an arbitrary set of nodes; those nodes in the set which are connected to other nodes not in the set are given concrete names and are called *attachment points*. The summary node `front` contains two attachment points `front::f` and `front::l` denoting the first and last elements of the `front` list respectively.

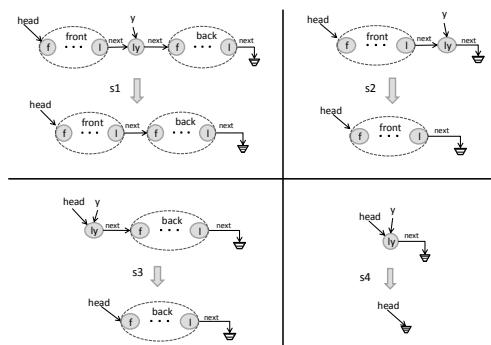
The state configurations are defined using a list of state predicates such as `(head -> front::f)` which denotes that the `head` variable points to the attachment point `f` of `front`. The other scenarios `s2`, `s3` and `s4` correspond to the cases of deleting the last node, the first node and the only node of the list respectively. Notice that there is no scenario corresponding to the case where the node to be deleted is not in the list. That means that the behavior of the synthesized code will be unspecified in such a case.

In order for the synthesizer to reason about summary nodes, the user needs to specify their structure. In this case, for example, the user needs to express the fact that `front` and `back` are not just arbitrary sets of nodes; they are lists. In SPT, this structural information is provided inductively through `unfold` rules. The two possible `unfold` rules for the summary node `front` and their corresponding visual description are shown in Figure 12. The rule states that the summary node `front` either represents a single node `x` or a node `x` followed by another similar summary node `front`. The `unfold` predicate consists of the summary node, the replacement node, the incoming and outgoing edges, and additional constraints that hold after the `unfold` operation. In SPT, the programmer can also provide `fold` rules to describe to the system how sets of nodes can be summarized by a single node. In most cases, such as the example, the synthesizer can reason about the correctness of a manipulation by using the inverse of the `unfold` rules for summarization, but for some tree algorithms, the synthesis process can be made more efficient by providing explicit `fold` rules for summarization.

```

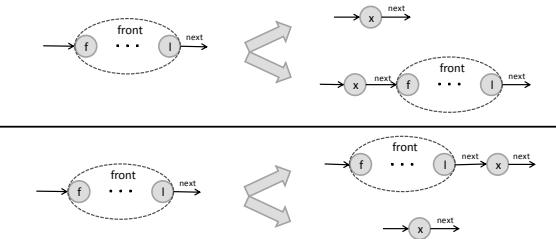
scenario s1
: head -> front::f, y -> ly
  front::l.next -> ly,
  back::l -> null, ly.next -> back::f
output: head -> front::f,
  front::l.next -> back::f,
  back::l.next -> null
scenario s2
: head -> front::f, y -> ly
  ly.next -> null, front::l.next -> ly
output: head -> front::f,
  front::l.next -> null
scenario s3
: head -> ly, ly.next -> back::f,
  y -> ly, back::l.next -> null
output: head -> back::f,
  back::l.next -> null
scenario s4
: head -> ly, ly.next -> null, y -> ly
output: head -> null

```



(a)

(b)

Fig. 1. Scenarios describing input and output state descriptions for linked list deletion

```

unfold front::f x [in (front::f, x)] [out (front::l, x)] ()
unfold front::f x [in (front::f, x)] [out (front::l, front::l)] (x.next -> front::f)

unfold front::f x [in (front::f, x)] [out (front::l, x)] ()
unfold front::f x [in (front::f, front::f)] [out (front::l, x)] (front::l.next -> x)

```

Fig. 2. Two possible unfold definitions for summary node front

In addition to the scenario descriptions, SPT also requires users to provide a loop skeleton of the desired implementation. This helps the synthesizer focus on implementations that are close to the user's expectations, and also lets them specify intermediate state constraints. The loop skeleton for the running example is shown in Figure 3(a). It consists of a while loop with a set of unknown statements before the loop, in the loop body and after the loop. The unknown statements are denoted by the ??(n) operator, where n represents the maximum length of the unknown statement block. SPT restricts unknown statements to be of two forms: i) guarded statements of the form if(**) then ASSIGN, where ** represents a conditional over pointer variables and ASSIGN denotes pointer assignments with at most one pointer dereference, and ii) unfold/fold statements of the form unfold var (resp. fold var) that corresponds to unfolding (folding) the location pointed to by variable var.

<pre>llDelete(Node head, Node y){ Node temp, prev; temp = head; while(temp != y){ // unfold temp1; prev = temp; temp = temp.next; // fold prev; } if(prev == null) head = temp.next; else prev.next = temp.next; }</pre> <p>(a)</p>	<pre>llDelete(Node head, Node y){ Node temp, prev; temp = head; while(temp != y){ // unfold temp1; prev = temp; temp = temp.next; // fold prev; } if(prev == null) head = temp.next; else prev.next = temp.next; } llinsert(Node head, Node y) { Node temp1, temp2; ??(2) /* h1 */ while(**){ /* h2 */ ??(4) /* h3 */ } /* position of y found */ yPosFound: ??(4) /* h4 */ }</pre> <p>(b)</p>	<pre>llinsert(Node head, Node y) { Node temp1, temp2; ??(2) /* h1 */ while(**){ /* h2 */ ??(4) /* h3 */ } /* position of y found */ yPosFound: ??(4) /* h4 */ }</pre> <p>(c)</p>
---	---	--

Fig. 3. (a) The loop skeleton and (b) the synthesized implementation for linked list deletion and (c) the loop skeleton for sorted linked list insertion

Given the scenarios, recursive definitions and the loop skeleton, SPT synthesizes the imperative implementation shown in Figure 3(b). The `true` conditionals and `skip` statements are removed from the code for better readability.

3 Algorithm

The details of the synthesis algorithm used by SPT can be found in [4]. At a high level, the algorithm first translates the loop skeleton into a set of equations relating the inputs and outputs of all the unknown blocks of code. Let F_i denote the unknown transfer function that maps a set of program states to another set of program states. The relationships between the inputs and outputs of all the transfer functions is captured by a set of equations of the form:

$$(t_0 = In_k) \wedge \forall_{v_i \in (V \setminus v_0)} t_i = F_i(\bigcup_{j \in pred(v_i)} t_j) \quad (1)$$

where $pred(v_i)$ denotes the predecessors of node v_i in the CFG. In_k denotes the input state constraint for the k^{th} scenario, and the goal is to find F_i such that $t_N = Out_k$ for each scenario k . The system works by representing the F_i as parameterized functions and solving for the parameters by using a counterexample guided inductive synthesis [4].

4 Tool Architecture

The architecture of Storyboard Programming Tool consists of four major components as shown in Figure 4.

A. Storyboard Parser: The parser takes as input a storyboard description (*.sb) written in the storyboard language and translates it into our intermediate constraint language that consists of a set of prolog predicates (*.pl). The

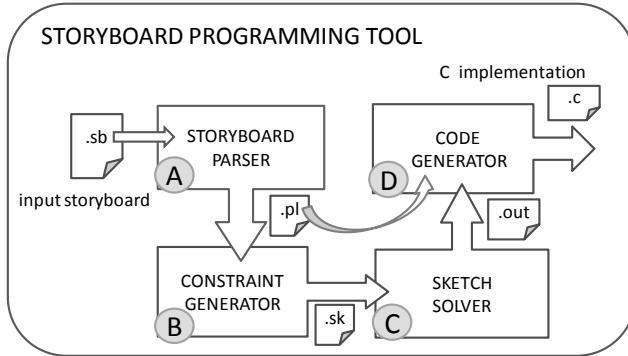


Fig. 4. The architecture of Storyboard Programming Tool

Storyboard language is more concise than [4] because the parser also performs some type inference to infer variables, locations, selectors, and summary nodes.

B. Constraint Generator: The constraint generator translates the Prolog predicates into a sketch constraint file (*.sk). SKETCH provides an expressive language to define the shape analysis problem, allowing us to use high-level language constructs such as functions, arrays and structures to model the abstract state, as well as holes to model the unknown transfer functions. The problem is encoded in a way that leverages the counterexample guided inductive synthesis algorithm of sketch to avoid having to represent sets of shapes explicitly.

C. Sketch Solver: The SKETCH solver solves the sketch constraint file and produces an output (.out) file where all unknown function choices are resolved.

D. Code Generator: The code generator takes as input the output generated by the SKETCH solver and completes the loop skeleton (.c) by mapping the function choice values to their corresponding program statements and conditionals using the intermediate constraint file (*.pl).

5 Experiments and Tool Experiences

We evaluated the tool on several linked list and binary search tree manipulations as well as AIG (And-Inverter Graph) insertion. The details about the experiments and benchmarks can be found in [4]. We present here a comparison with the SKETCH tool on a small sample of benchmarks.

Comparison with SKETCH: Table 1 shows the running times of the Storyboard tool with the SKETCH system. We can see that SKETCH is faster, but can only perform bounded reasoning ($N=5$) and quickly times out for larger values of N . On the other hand, the storyboard tool performs unbounded analysis using abstract interpretation. However, the biggest difference we found was in the usability of the tools. We had to spend almost three hours for writing a spec in SKETCH for these manipulations. For writing a sketch, one has to write a converter (and its inverse) for converting (resp. translating back) an array to that data structure. Then one has to use quantified input variables for writing tricky

Table 1. Performance comparison with Sketch

Benchmark	Sketch (N=5)	Storyboard
ll-reverse	18s	1m40s
ll-insert	31s	2m3s
ll-delete	25s	2m8s
bst-search	39s	2m51s
bst-insert	3m35s	3m12s

specs. In our storyboard tool, we only have to provide input-output examples which in our experience was a lot more natural. We now present some of our other experiences in handling complicated manipulations with the tool.

Intermediate State Configurations: Our tool allows users to write intermediate state configurations to reduce the search space and enable the synthesizer to synthesize more complex manipulations. The user can label a program location in the loop skeleton and provide the state description at that point using the `intermediate` keyword as part of the scenario description. For example in the case of insertion in a sorted linked list, we add an additional insight based on the fact that the loop skeleton for the insertion (Figure 3(c)) is performing two tasks: first to find a suitable location for inserting the node y and the second task of inserting y into the list. In the abstract scenario, we provide an intermediate state configuration at the label `yPosFound` in which two variables point to the two locations between which the insertion is to be performed.

The SPT tool illustrates a new approach to synthesis namely *Multimodal Synthesis*, where the synthesizer takes input specification in many different forms such as concrete examples, abstract examples, and implementation insights, and synthesizes code that is provably consistent with all of them. We believe this idea of multimodal synthesis has applicability in many other domains as well.

References

1. Jobstmann, B., Griesmayer, A., Bloem, R.: Program Repair as a Game. In: Etesami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
2. Lev-Ami, T., Sagiv, M.: TVLA: A System for Implementing Static Analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)
3. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118 (1999)
4. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: FSE, pp. 289–299 (2011)
5. Solar-Lezama, A.: Program Synthesis By Sketching. PhD thesis, EECS, UC Berkeley (2008)
6. Solar-Lezama, A., Rabbah, R., Bodik, R., Ebciooglu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294 (2005)
7. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010)
8. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL, pp. 327–338 (2010)

CSolve: Verifying C with Liquid Types*

Patrick Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala

University of California, San Diego
`{prondon,abakst,mwookawa,jhala}@cs.ucsd.edu`

Abstract. We present CSOLVE, an automated verifier for C programs based on Liquid Type inference. We show how CSOLVE verifies memory safety through an example and describe its architecture and interface.

1 Introduction

Verifying low-level programs is challenging due to the presence of mutable state, pointer arithmetic, and unbounded heap-allocated data structures. In recent years, dependent refinement types have emerged as a promising approach to verification in general [1,7] and low-level software in particular [2]. In a refinement type system, each program variable and expression is given a type of the form $\{\nu : \tau \mid p\}$ where τ is a conventional type such as `int` or `bool` and p is a logical predicate over the program variables describing the values ν which belong to the type, called the *refinement predicate*. To keep type checking decidable, refinement predicates are typically drawn from a quantifier-free logic; by combining SMT-based logical reasoning and type theory-based data structure reasoning, refinement type systems are easily able to synthesize and reason using facts about the contents of unbounded data structures.

While powerful, refinement types have typically been associated with a high annotation burden on the programmer. We present CSOLVE, an automated verifier for C programs based on the Low-Level Liquid Types [6] technique for refinement type inference. We show how CSOLVE accommodates refinement type checking with little necessary annotation.

2 Architecture, Use, and Availability

Type inference in CSOLVE is split into four phases. In the first phase, the input C program is read by CIL [4], which generates an AST. This AST is then simplified in various ways, the most significant of which is that the code is transformed to SSA so that local variables are never mutated. The second phase generates physical types for each declared function and global variable and checks that the program code respects these types. The third phase walks the CIL AST and assigns each expression and variable in the program a refinement type with

* This work was supported by NSF grants CCF-0644361, CNS-0720802, CCF-0702603, and a gift from Microsoft Research.

a distinct *refinement variable* representing its as-yet-unknown refinement predicate. The same phase generates subtyping constraints over these refinement types such that solving for the refinement variables within the constraints yields a valid typing for the program. The fourth phase attempts to solve the subtyping constraints using a fixed-point procedure based on predicate abstraction, using the Z3 SMT solver [3] to discharge the logical validity queries that arise in constraint solving.

Input. CSOLVE takes as input a C source code file and a file specifying the set of logical predicates to use in refinement inference. Predicates are also read from a standard library of predicates that have proven to be useful on a large variety of programs, further easing the programmer’s annotation burden.

Output. If the program is type-safe, CSOLVE outputs “Safe”. Otherwise, the program may be type-unsafe, according to either the physical type system or the refinement type system. In either case, for each error, CSOLVE prints the name of the file and line number where the error occurs, as well as a description of the error. In the case where the error is in refinement type inference, CSOLVE prints the subtyping constraints which cannot be solved. Whether the program typechecks or not, CSOLVE produces a mapping of program identifiers to their inferred types which can be viewed using the tag browsing facilities provided by common editors, *e.g.* Vim and Emacs.

Compatibility With C Infrastructure. Thanks to the infrastructure provided by CIL, CSOLVE is able to work as a drop-in replacement for GCC. Hence, to check a multi-file program one need only construct or slightly modify a makefile which builds the program from source.

Availability. The CSOLVE source code and an online demo are available at <http://goto.ucsd.edu/csolve>.

3 Example

In the following, we demonstrate the use of CSOLVE through a series of functions which manipulate text containing comma-separated values. We begin by showing how CSOLVE typechecks library functions against their stated specifications. We then show how CSOLVE infers function types to check an entire program.

We begin with a string library function, `strtolower`, shown in Figure 1, which lowerscases each letter in a string. Its type signature is a C type augmented with annotations that are used by the CSOLVE typechecker. The `CHECK_TYPE` annotation tells CSOLVE to check `strtolower` against its type signature, rather than attempting to infer its type from its uses.

Type checking `strtolower` proceeds in two phases. First, because C is untyped, a *physical type checking* pass recovers type information describing heap layout and the targets of pointer-valued expressions. Next, a *refinement type checking* pass computes refinement types for all local variables and expressions.

Physical Type Checking. We begin by describing how the type annotations in the example are used by CSOLVE to infer enriched physical types. The

```

void strntolower (char * STRINGPTR SIZE_GE(n) s,
                  int NONNEG n)
CHECK_TYPE {
    for (; n-- && *s != '\0'; s++)
        *s = tolower ((*s));
}
extern char * NNSTRINGPTR LOC(L)
NNREF([& [s <= v; v < s + n; InB(s)]])
strnchr (char * STRINGPTR LOC(L) SIZE_GE(n) s,
          int NONNEG n,
          char c);
typedef struct _csval {
    int len;
    char * ARRAY LOC(L) str;
    struct _csval * next;
} csval;
void lowercase_csvals (csval *v) {
    while (v) {
        strntolower (v->str, v->len);
        v = v->next;
    }
}
csval INST(L, S) *
revstrncsvals (char * ARRAY LOC(S) s,
                int n)
{
    csval *last = NULL;
    while (n > 0) {
        csval *v =
            (csval *) malloc (sizeof (*v));
        v->next = last;
        v->str = s;
        char *c = strnchr (s, n, '.');
        if (!c) c = s + n - 1;
        *c = '\0';
        v->len = c - s;
        n -= v->len + 1;
        s = c + 1;
        last = v;
    }
    return last;
}
...
1. csval *vs =
    revstrncsvals (line, len);
2. lowercase_csvals (vs);
...

```

Fig. 1. Running example: splitting a string into comma-separated values

`char * STRINGPTR` portion of the type ascribed to `strntolower`'s parameter `s` indicates to CSOLVE that `s` is a reference to a location `l` which contains an array of characters, *i.e.*, a string (and not a single `char`). Concretely, the type of `s` is `ref(l, {0 + 1*})`, which indicates that `s` points into a region of memory named by `l`. The notation `{0 + 1*}`, which is equivalent to `{0, 1, 2, ...}`, indicates that `s` may point to any nonnegative offset from the start of the region `l`, *i.e.*, anywhere in the array. Based on `s`'s type, CSOLVE describes the heap as

$$l \mapsto \{0 + 1*\} : \text{int}(1, \{0 \pm 1*\}).$$

The above heap contains a single location, `l`, whose elements are offsets from `l` in the set `{0 + 1*}`, defined as above. Each array element has the type `int(1, {0 ± 1*})`, which is the type of one-byte integers (`chars`) whose values are in the set `{..., -1, 0, 1, ...}` (*i.e.*, any `char`). Similarly, the physical type of `n` is `int(4, {0 ± 1*})`.

CSOLVE then determines, through straightforward abstract interpretation in a domain of approximate integer values and pointer offsets [6], that the physical types of `s`, `n`, and the heap are preserved by the loop within the body of `strntolower`; we note only that the return type of `tolower` indicates that it returns an arbitrary `char`, as above. Thus, physical typechecking succeeds, and we proceed to refinement type checking.

Refinement Type Checking. We next explain how CSOLVE typechecks the body of `strntolower`—in particular, to verify that `strntolower`'s type signature implies the safety of the array accesses in its body.

We begin by describing how the annotations ascribed to `strntolower` are translated to a refinement type by CSOLVE. The type of `s` uses the convenience macros `STRINGPTR` and `SIZE_GE`, defined as:

$$\begin{aligned} \text{STRINGPTR} &= \text{ARRAY REF}(\text{SAFE}(\nu)) \\ \text{SIZE_GE}(n) &\doteq \text{REF}(BE(\nu) - \nu \geq n). \end{aligned}$$

In the above, `ARRAY` indicates that the refined type points to an array, used in physical type checking. The `REF` macro is used to attach a refinement predicate to a type. Refinement predicates can themselves be constructed using macros; `SAFE` is a macro defined as the predicate

$$\text{SAFE}(p) \doteq 0 < p \wedge BS(p) \leq p \wedge p < BE(p).$$

In this definition, the functions `BS(p)` and `BE(p)` indicate the beginning and end of the memory region assigned to pointer `p`, respectively. Thus, the `SAFE(p)` predicate states that `p` is a non-NULL pointer that points within the memory region allocated to `p`, *i.e.*, `p` is within bounds. The predicate `SIZE_GE(n)` states that the decorated pointer points to a region containing at least `n` bytes; note that this expresses a *dependency* between the type of `s` and the value of the parameter `n`. We decorate the type of `n` with `NONNEG`, which expands to `REF($\nu \geq 0$)`.

We now describe how CSOLVE uses the given types for `s` and `n` to verify the safety of `strntolower`. To do so, CSOLVE infers *liquid types* [5] for the variables `s` and `i` within the body of `strntolower`, as well as the contents of the heap. A liquid type is a refinement type whose refinement predicate is a conjunction of user-provided *logical qualifiers*. Logical qualifiers are logical predicates ranging over the program variables, the wildcard \star , which CSOLVE instantiates with the names of program variables, and the *value variable* ν , which stands for the value being described by the refinement type. Below, we assume the logical qualifiers

$$\begin{aligned} Q_0 &\doteq \{0 \leq \nu, \text{SAFE}(\nu), \star \leq \nu, \nu + \star \leq BE(\nu), \nu \neq 0 \Rightarrow \text{InB}(\nu, \star)\} \\ \text{where } \text{InB}(p, q) &\doteq BS(p) = BS(q) \wedge BE(p) = BE(q) \end{aligned}$$

where `InB(p, q)` means `p` and `q` point into the same region of memory.

From the form of the loop and the given type for the parameter `n`, CSOLVE infers that, within the body of the loop, `n` has the liquid type

$$n :: \{\nu : \text{int}(4, \{0 \pm 1\}) \mid 0 \leq \nu\}.$$

Based on the type given for the parameter `s` and the form of the loop, CSOLVE infers that, within the loop, `s` has the liquid type

$$s :: \{\nu : \text{ref}(l, \{0 + 1\}) \mid s \leq \nu \wedge \nu + n \leq BE(\nu) \wedge \nu \neq 0 \Rightarrow \text{InB}(\nu, s)\}.$$

The predicates $s \leq \nu$, $\nu + n \leq BE(\nu)$, and $\nu \neq 0 \Rightarrow \text{InB}(\nu, s)$ are instantiations of qualifiers $\star \leq \nu$, $\nu + \star \leq BE(\nu)$, and $\nu \neq 0 \Rightarrow \text{InB}(\nu, \star)$ from \mathbb{Q}_0 , respectively, where the \star has been instantiated with s , n , and s , respectively. By using an SMT solver to check implication, CSOLVE can then prove that s has the type

```
s :: { $\nu : \text{ref}(l, \{0 + 1^*\}) \mid \text{SAFE}(s)\}$ }
```

and thus that the accesses to $*s$ within `strntolower` are within bounds.

External Definitions. If the user specifies a type for a function with the `extern` keyword, CSOLVE will use the provided type when checking the current source file, allowing the user to omit the body of the external function. This allows for modular type checking and, by abstracting away the details of other source files, it permits the user to work around cases where a function may be too complex for CSOLVE to typecheck.

In the sequel, we use the library function `strnchr`, which attempts to find a character c within the first n bytes of string s , returning a pointer to the character within s on success and `NULL` otherwise. Its type, declared in Figure 1, illustrates two new features of CSOLVE’s type annotation language. First, macros that begin with `NN` are analogous to the versions without the `NN` prefix, but the refinement predicates they represent are guarded with the predicate ($\nu \neq 0$). Such macros are used to indicate properties that are only true of a pointer when it is not `NULL`. Second, the annotation `LOC(L)` is used to provide may-aliasing information to CSOLVE. The annotation `LOC(L)` on both the input and output pointers of `strnchr` indicates that both point to locations in the same may-alias set of locations, named L . This annotation is necessary because CSOLVE assumes by default that all pointers passed into or out of a function refer to distinct heap locations. This assumption that pointers do not alias is *checked*: if the annotation were not given, CSOLVE would alert the user that locations that were assumed distinct may become aliased within the body of `strnchr`.

Whole-Program Type Inference. The remainder of Figure 1 shows a fragment of a program which reads lines of comma-separated values from the user, splits each line into individual values (`revstrncsvs`), and then transforms each value to lowercase (`lowercase_csvs`). In the following, we describe how CSOLVE performs refinement type inference over the whole program to determine that all of its memory accesses are safe.

The remainder of the program manipulates linked lists of comma-separated values, described by the structure type `csval`. Note that the field `str` is annotated with the `ARRAY` attribute, as before, as well as a may-alias set, L . By declaring that the `str` field points to may-alias set L , we *parameterize* the `csval` structure by the location set L that its `str` field points into. The programmer can then instantiate the parameterized location set according to context to indicate potential aliasing between the `str` field and other pointers. For example, the annotation `INST(L, S)` in the type of `revstrncsvs` instantiates `csval`’s may-alias set parameter L in that type to the location set S , indicating that the input string s and the strings stored in the list of `csvals` reside in heap locations in the same may-alias set, S .

In the following, we assume the set of qualifiers is

$$\begin{aligned} \mathbb{Q} = \mathbb{Q}_0 \cup \{ & \nu \neq 0 \Rightarrow \star \leq \nu, \nu \neq 0 \Rightarrow \nu < \star + \star, \nu \neq 0 \Rightarrow \nu = BS(\nu), \\ & \nu \neq 0 \Rightarrow BE(\nu) - BS(\nu) = 12, \nu = \star + (\star - \star) \}. \end{aligned}$$

At the line marked 1, we assume CSOLVE has inferred the types

$$\begin{aligned} \text{line}::\{ & \nu : \text{ref}(l, \{0 + 1*\}) \mid \text{SAFE}(\nu) \wedge \text{SIZE_GE}(\text{len}) \} \\ \text{len}::\{ & \nu : \text{int}(4, \{0 \pm 1*\}) \mid \text{true} \}. \end{aligned}$$

From line 1, CSOLVE infers that argument **s** of `revstrncsvs` of has type

$$s::\{\nu : \text{ref}(l, \{0 + 1*\}) \mid \text{SAFE}(\nu) \wedge \text{SIZE_GE}(n)\}.$$

CSOLVE infers that this type is a loop invariant, and thus that the call to `strncpy` is type-correct. CSOLVE infers from the type of `malloc` that **v** has type

$$\begin{aligned} v::\{ & \nu : \text{ref}(l_v, \{0\}) \mid \text{VALPTR}(\nu) \} \\ \text{VALPTR}(p) = & \nu = BS(\nu) \wedge BE(\nu) - BS(\nu) = 12 \wedge \nu > 0, \end{aligned}$$

indicating that **v** is a non-NULL pointer to a 12-byte allocated region; this allows CSOLVE to verify the safety of the indirect field accesses.

Finally, CSOLVE infers that **v** and **last** refer to elements within a set of run-time locations, collectively named l_v . Each location in l_v has type

$$\begin{aligned} l_v \mapsto 0 : \{ & \nu : \text{int}(4, \{0 \pm 1*\}) \mid 0 \leq \nu \}, \\ 4 : \{ & \nu : \text{ref}(l, \{0 + 1*\}) \mid \text{SAFE}(\nu) \wedge \text{SIZE_GE}(@0) \}, \\ 8 : \{ & \nu : \text{ref}(l_v, \{0\}) \mid \nu \neq 0 \Rightarrow \text{VALPTR}(\nu) \}. \end{aligned}$$

Offsets 0, 4, and 8 in the type of l_v correspond to the fields **len**, **str**, and **next**, respectively. The notation $@n$ is used in refinement predicates to refer to the value stored at offset n within the location; in this case, $@0$ is used to indicate that the **str** field points to an allocated region of memory whose size is at least the value given in the **len** field. The type of heap location l is as given earlier. The type of the **next** field indicates that it contains a pointer to the location l_v , *i.e.*, that the **next** field contains a pointer to the same kind of structure. Thus, CSOLVE that lists constructed by `revstrncsvs` satisfy the above invariant.

Because the pointer **last** is returned from `revstrncsvs` and due to the call on line 2, as well as the type of l_v given above, CSOLVE is able to determine that the array accesses and call to `strntolower` within `lowercase_csvs` are safe, and thus prove the program is memory safe.

References

1. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffei, S.: Refinement types for secure implementations. In: CSF (2008)
2. Condit, J., Harren, M., Anderson, Z., Gay, D.M., Necula, G.C.: Dependent Types for Low-Level Programming. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 520–535. Springer, Heidelberg (2007)
3. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
5. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
6. Rondon, P., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL, pp. 131–144 (2010)
7. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL (1999)

PASSERT: A Tool for Debugging Parallel Programs

Daniel Schwartz-Narbonne*, Feng Liu, David August, and Sharad Malik*

Princeton University
`{dstwo,fengliu,august,sharad}@princeton.edu`

Abstract. PASSERT is a new debugging tool for parallel programs which allows programmers to express correctness criteria using a simple, expressive assertion language. We demonstrate how these *parallel assertions* allow the detection and diagnosis of real world concurrency bugs, detecting 14/17 bugs in an independently selected set of bugs from open source software. We describe a runtime checker which allows automatic checking of parallel assertions in C and C++ programs, with a geometric mean of 6.6 \times overhead on a set of PARSEC benchmarks. We improve performance by introducing a *relaxed timing semantics* for parallel assertions, which better reflects real memory models, and exposes more bugs with less overhead (geometric mean overhead 3.5 \times).

1 Introduction

PASSERT is a new debugging tool for parallel programs which allows programmers to express correctness criteria using a simple, expressive assertion language. If a correctness property is violated during program execution, an automatic runtime checker will detect the violation.

Such a tool is necessary because the standard assertions that are widely used for debugging sequential programs are highly limited for parallel programming. In a sequential program, it is sufficient to check whether a property holds at a particular point in time. In a parallel program it is possible for a property to be true when a section of code begins executing, for the code in question to make no changes that could falsify the property, and yet for the property to be violated by the actions of a second thread. Checking whether a property holds during execution through a small code segment potentially requires annotating every statement of the program with assertions. Even this might not be sufficient: since assertions are not synchronized with code execution, checking an assertion that depends on more than one program variable might be impossible to do correctly without significant code modification.

Parallel assertions, the input language of PASSERT, solve this problem by providing a simple, understandable set of predicates that allow programmers to write local assertions which allow testing of multithreaded programs.

* The authors acknowledge the support of the Gigascale Systems Research Center (GSRC), one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation (SRC) entity.

2 The Parallel Assertion Language

2.1 Syntax

A programmer debugging a piece of code wants to know whether a property holds during the execution of a piece of code. A parallel assertion therefore consists of two parts: a description of the times when the assertion is expected to be true, which we call the *assertion scope*, and the property which is expected to hold, which we call the *assertion condition*. Parallel assertions are expressed using simple, easy to use predicates. More powerful formulations, such as temporal logic, are possible but are unfamiliar to programmers and provide expressiveness at the cost of complexity.

Parallel assertions are applicable to many programming languages. This paper focuses on our C/C++ implementation; a full formal syntax and semantics is provided in [3].

Assertion Scope. The assertion scope is a block of code which delineates the time during which the assertion condition must hold. It begins with the keyword `thru` and ends with `passert(cond)`.

```
thru {
    ...
} passert (cond)
```

Assertion Condition. The assertion condition is a side-effect free Boolean expression, which can contain Boolean combinations of any of the following sub-expressions:

Type	Description	Example
Value	Any side effect free boolean expression	$x > 5$
LocalRead(x)	True when the asserting thread is reading the variable x	LR(x)
LocalWrite(x)	True when the asserting thread is writing the variable x	LW(x)
RemoteRead(x)	True when a thread other than the asserting thread is reading the variable x	RR(x)
RemoteWrite(x)	True when a thread other than the asserting thread is writing the variable x	RW(x)
HasOccurred(expr)	True iff expr has ever been true while the assertion was active	HO(expr)

2.2 Assertion Semantics

The result of a parallel assertion is defined relative to a program execution i.e. an observed interleaving of read and write events by the executing threads. We augment this program execution by adding *assertion begin* and *assertion end* events, which mark the beginning and end of assertion scopes. A *timeline* is an observed total ordering of these events for a particular execution of a parallel program. (Under certain circumstances, discussed in Sec. 6.2, this requirement for a total ordering can be relaxed).

An assertion holds, for a particular timeline, if the assertion condition is true for all times between the beginning and end assertion events. It fails if there is any time between the begin and end events during which the assertion condition is not true. Since assertions are checked, not enforced, they can be used for debugging and then turned off for production.

3 Applicability of Parallel Assertions to Real World Bugs

Assertions have two purposes: to detect unexpected events that may represent bugs, and to test hypotheses to diagnose the cause of these bugs. We evaluated the effectiveness of parallel assertions using the University of Michigan Collection of Concurrency Bugs [4]¹, an independently selected collection of real world concurrency bugs from major open source programs. For each bug, we attempted to write a parallel assertion to detect its symptoms and diagnose its underlying cause, without requiring any other code modifications.

Bug Analysis Example MySQL-3.23.56 had a concurrency bug which caused it to produce a nonsensical log: for example, it could report a successful insert before the associated table had been created. We identified several possible explanations for this bug, and wrote a parallel assertion to test each of these.

- Assertion 1 checked for a data-race problem and confirmed that all accesses to the log are protected by a lock.
- Assertion 2 determined that inserts never occurred while the table was invalid.
- Assertion 3 tested whether the log order represents the actual order of events, i.e. does an operation (such as creating a table), and the logging of that operation, form a single atomic unit. This seemingly simple test requires the expressiveness of parallel assertions. It would be incorrect to mark the entire `generate_table()` function as atomic, because it correctly accesses shared variables in a non-transactional way. In addition, while conflicting writes to the logger represent an error, reads may not. This assertion captures these subtleties, and successfully diagnosed the cause of the error.

These assertions could subsequently be left in the program as regression tests.

<pre>int generate_table(...) { ... thru{ pthread_mutex_lock(&LOCK_open); // delete the original table // create a new table pthread_mutex_unlock(&LOCK_open); mysql_update_log.write(...); }passert(!RW(mysql_update_log)); ... }</pre>	<pre>bool MYSQL_LOG::write(...) { ... pthread_mutex_lock(&LOCK_log); // log event pthread_mutex_unlock(&LOCK_log); ... }</pre>
---	--

(Parallel assertion to identify MySql bug 169)

¹ Currently maintained at <http://www.eecs.umich.edu/~jieyu/bugs.html>

Summary of Bug Coverage Almost all (14/17) of the bugs in the University of Michigan Collection can be detected using parallel assertions (shown in the table below). Of these, thirteen can also be diagnosed using parallel assertions; one is a multi-function atomicity violation, which would be difficult to capture in a single syntactic scope. PASSERT is not designed to detect deadlock and complex order-violation bugs.

Bug ID	Bug Type	Detect Symptom	Diagnose Cause
Apache #25520, 21287			
MySQL #44, 791, 2011, 3596, 12848	Data Race	Yes	Yes
Cherokee Bug1, Aget Bug2			
Apache #45605			
MySQL #169, 12228	Atomicity	Yes	Yes
Memcached #127			
Apache #21285	Atomicity	Yes	No
Pbzip2 0.9.4, Transmission 1.42	Order Violation	No	No
Aget Bug1	Deadlock	No	No

4 Tool Design

PASSERT is a compiler that automatically adds runtime support for parallel assertions in C/C++ programs. It is implemented as an extension to the LLVM [2] compiler suite, and supports the same programs and language features as the standard LLVM compiler. At present, PASSERT targets programs using the *pthreads* threading library; we expect that it will be easy to extend it to other threading models, such as Windows threads.

We reduced the impact of assertion evaluation on program execution by decoupling execution and checking: as the program executes, relevant loads/stores are timestamped and logged for subsequent checking.

Logging. To reduce logging overhead, we use alias analysis to determine whether a load or store may access a variable in a parallel assertion condition. Since static alias analysis is overly conservative, PASSERT also maintains a hash-table which records whether accesses to a given memory location need to be logged. Collisions in the hash-table may cause unnecessary logging, but will never cause an event to be missed from the log.

Checking. Checking can either be done online, using a separate checking thread and synchronized queues, or offline, in which case no synchronization needs to be done on the queues. Performance results are discussed in Sec. [7].

Avoiding Stalls. An event can only be processed if all events which occur before it in the execution trace have already been processed. If a thread stops

generating events, it becomes impossible to determine the correct sequence of events, since the checker has no way of distinguishing between “no event” and “a not-yet-logged event”. The checker must therefore conservatively wait until the thread resumes generating events. If a thread which is about to stall can generate a *Thread Stalled* event, the checker can continue without waiting. PASSERT automatically generates such events before calls to blocking functions such as `pthread_join()`, `pthread_barrier_wait()` and `pthread_cond_wait()`. Programmers writing specialized synchronization libraries can add their own event annotations. They can also insert *Heartbeat Events* into code which is unlikely to generate any logged events, such as calculations on privatized data. As a future extension, we hope to introduce heuristics that will automatically add these events at appropriate points.

5 Response to Assertion Failure

When an assertion fails, PASSERT informs the user and prints out a set of diagnostic information. This information includes which memory access caused the assertion failure (including time, `thread_id`, value, and type of access), as well as which assertion was triggered. If the user desires, PASSERT can output its full log to a file. If the program has been compiled with debug symbols, it is possible to associate the log information with program locations, although this is not currently implemented. A compiler flag controls whether the executable should abort or continue after an assertion violation.

In addition, PASSERT provides a feedback function which allows user code to block until the checker has evaluated all events before the feedback function call began, and then returns the checker status (i.e. failure or success). The program can use this mechanism to ensure that a dangerous action only occurs after correct execution, or to rollback and recover after an assertion failure. As a convenience, PASSERT can automatically insert a checker feedback call at the end of each `thru` block.

6 Timing modes

The semantics of parallel assertions, as introduced in [3], requires a total ordering of events during a concurrent execution. However, modern microprocessors typically have more relaxed semantics, which allow for event sequences which do not have any consistent total order.

6.1 Strict Timing

In *strict timing mode*, this total ordering is enforced through the use of locks and fences around every logged memory access. Timestamps can be acquired either through a global counter, or through a hardware timestamping mechanism such as RDTSC [1].

6.2 Relaxed Timing

Not all parallel assertions require a total order over program events in order to be correctly evaluated. In some cases, a partial order among certain types of events is sufficient. In particular, any assertion which either:

- Only contains access predicates (such as `RemoteWrite(x)`), or
- Contains value predicates, but only references a single variable

requires a partial order between access and assertion begin/end events, but does not require any further ordering among access events.

Relaxed Timing Mode enforces only these minimal constraints, dramatically reducing the number of locks and memory fences required. This both reduces runtime overhead (see Sec. 7), and allows a wider range of bugs to manifest, since locks prevent certain combinations and orderings of events that would be legal and possible in the underlying hardware model.

7 Results

We evaluated the runtime performance of PASSERT using the assertion-annotated PARSEC benchmarks described in [3]. All benchmarks were compiled at optimization level O3, and were executed on a quad core Intel X3440 with 16GB of RAM. The runtime for each benchmark, normalized to the unmodified benchmark compiled using standard gcc, is reported in Fig. 1.

The online checker performs checking in parallel with execution, which speeds up the checking phase, but requires extra synchronization in the logging phase. Currently, these two effects roughly cancel each other out; we hope to remove this overhead with further optimization. Strict timing had a geometric mean overhead of $6.6\times$. Relaxed timing was significantly faster, with a geometric mean overhead of $3.5\times$.

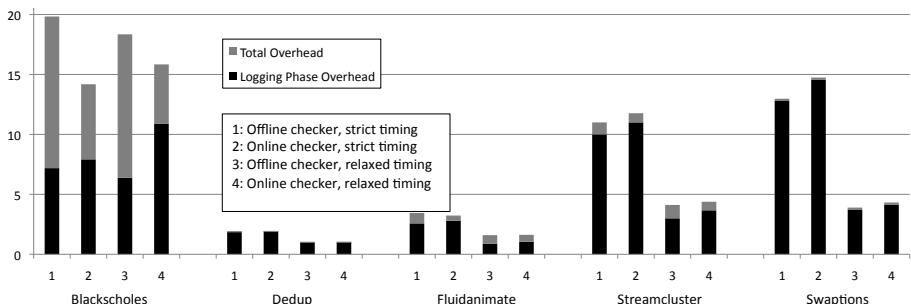


Fig. 1. Runtime overhead for parsec benchmarks

8 Conclusion

PASSERT provides programmers with a new tool to identify elusive bugs in parallel programs. Until now, parallel programs have been challenging to debug, because it has been hard to express and check assumptions about program execution across multiple threads. Our experience with the Michigan Bug Collection shows that parallel assertions are sufficiently expressive to capture a range of real-world bugs. Our performance experiments indicate that checking these assertions can be done with reasonable overhead. The simple, expressive syntax of PASSERT allows programmers to express correctness conditions to debug programs with a high degree of efficacy and a minimum of effort.

References

1. Intel Corporation. Intel 64 and IA-32 Architectures Developer’s Manual (2010)
2. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002)
3. Schwartz-Narbonne, D., Liu, F., Pondicherry, T., August, D., Malik, S.: Parallel assertions for debugging parallel programs. In: MEMOCODE 2011, pp. 181–190 (2011)
4. Yu, J., Narayanasamy, S.: A case for an interleaving constrained shared-memory multi-processor. In: ISCA 2009, pp. 325–336 (2009)

TRACER: A Symbolic Execution Tool for Verification*

Joxan Jaffar¹, Vijayaraghavan Murali¹, Jorge A. Navas², and Andrew E. Santosa³

¹ National University of Singapore

² The University of Melbourne

³ University of Sydney

Abstract. We present TRACER, a verifier for safety properties of sequential C programs. It is based on symbolic execution (SE) and its unique features are in how it makes SE finite in presence of unbounded loops and its use of interpolants from infeasible paths to tackle the *path-explosion* problem.

1 Introduction

Recently *symbolic execution* (SE) [15] has been successfully proven to be an alternative to CEGAR for program verification offering the following benefits among others [12][18]: (1) it does not explore infeasible paths avoiding expensive refinements, (2) it avoids expensive *predicate image* computations (e.g., *Cartesian* and *Boolean* abstractions [2]), and (3) it can recover from *too-specific* abstractions as opposed to monotonic refinement schemes often used. Unfortunately, it poses its own challenges: (C1) exponential number of paths, and (C2) infinite-length paths in presence of unbounded loops.

We present TRACER, a SE-based verification tool for *finite-state* safety properties of sequential C programs. Informally, TRACER attempts at building a finite symbolic execution tree which overapproximates the set of all concrete reachable states. If the error location cannot be reached from any symbolic path then the program is reported as safe. Otherwise, either the program may contain a bug or it may not terminate. The most innovative features of TRACER stem from how it tackles (C1) and (C2).

In this paper, we describe the main ideas behind TRACER and its implementation as well as our experience in running real benchmarks.

1.1 State-Of-The-Art Interpolation-Based Verification Tools

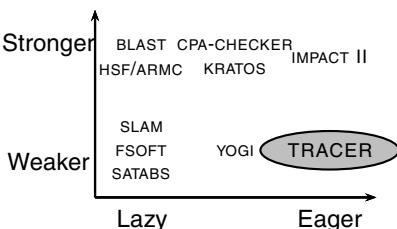


Fig. 1. State-of-the-art verifiers

Fig. 1 depicts one possible view of current verification tools based on two dimensions: *laziness* and *interpolation strength*. *Lazy* means that the tool starts from a coarsely abstracted model and then refines it while *eager* is its dual, starting with the concrete model and then removing irrelevant facts. CEGAR-based tools [14][7][10][21] are the best examples

* This paper extends the ideas published in [12][13] by describing a method for computing weakest preconditions as interpolants as well as a detailed description of the architecture of the tool and a new experimental evaluation.

of lazy approaches while SE-based tools [12][18] are for eager methods. Special mention is required for hybrid approaches such as YOGI [20], CPA-CHECKER [3], and KRATOS [5]. YOGI computes weakest preconditions from symbolic execution of paths as a cheap refinement for CEGAR. One disadvantage is that it cannot recover from too-specific refinements (see program *diamond* in [18]). CPA-CHECKER and KRATOS encode loop-free blocks into Boolean formulas that are then subjected to an SMT solver in order to exploit its (learning) capabilities and avoid refinements due to coarser abstractions often used in CEGAR. On the other hand, the performance of interpolation-based verifiers depends on the logical strength of the interpolants¹. In lazy approaches, a weak interpolant may contain spurious errors and cause refinements too often. Stronger interpolants may delay convergence to a fixed point. In eager approaches, weaker interpolants may be better (e.g., for loop-free fragments) than stronger ones since they allow removing more irrelevant facts from the concrete model.

TRACER performs SE computing efficient approximated *weakest preconditions* as interpolants. To the best of our knowledge, TRACER is the first publicly available (paella.d1.comp.nus.edu.sg/tracer) verifier with these characteristics.

2 How TRACER Works

Essentially, TRACER implements classical symbolic execution [15] with some novel features that we will outline along this section. It takes symbolic inputs rather than actual data and executes the program considering those symbolic inputs. During the execution of a path all its constraints are accumulated in a first-order logic (FOL) formula called *path condition* (*PC*). Whenever code of the form *if(C) then S1 else S2* is reached the execution forks the current symbolic state and updates path conditions along both the paths: $PC_1 \equiv PC \wedge C$ and $PC_2 \equiv PC \wedge \neg C$. Then, it checks if either PC_1 or PC_2 is unsatisfiable. If yes, then the path is *infeasible* and the execution halts backtracking to the last choice point. Otherwise, it follows the path. The verification problem consists of building a *finite* symbolic execution tree that overapproximates all concrete reachable states and proving for every symbolic path the error location is unreachable.

The first key aspect of TRACER, originally proposed in [13] for symbolic execution, is the avoidance of full enumeration of symbolic paths by *learning* from infeasible paths computing *interpolants* [8]. Preliminary versions of TRACER [12][13] computed interpolants based on *strongest postconditions* (*sp*). Given two formulas *A* (symbolic path) and *B* (last guard where infeasibility is detected) such that $A \wedge B$ is unsat, an interpolant was obtained by $\exists \bar{x} \cdot A$ where \bar{x} are *A*-local variables (i.e., variables occurring only in *A*). However, unlike CEGAR, TRACER starts from the concrete model of the program and then deletes irrelevant facts. Therefore, the weaker the interpolant is the more likely it is for TRACER to avoid exploring other “similar” symbolic paths. This is the motivation behind an interpolation method based on *weakest preconditions* (*wp*).

Example 1. The verification of the contrived program in Fig. 2(a) illustrates the need for *wp* as well as the essence of our approach to mitigate the “path-explosion” problem. Fig. 2(b) shows the first symbolic path explored by TRACER which is infeasible. (*)

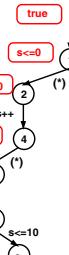
¹ Given formulas *A* and *B* such that $A \wedge B$ is unsatisfiable, a *Craig interpolant* [8] *I* satisfies: (1) $A \models I$, (2) $I \wedge B$ is unsatisfiable, and (3) its variables are common to *A* and *B*. We say an interpolant *I* is stronger (weaker) than *I'* if $I \models I'$ ($I' \models I$).

```

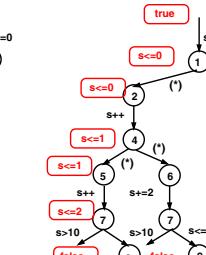
<0> s=0;
<1> if(*)
<2>   s++;
else
<3>   s+=2;
<4>   if(*)
<5>     s++;
else
<6>     s+=2;
<7>   if(s > 10)
<8>     error();
<9>

```

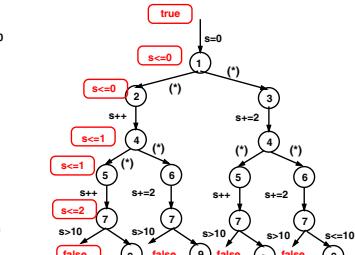
(a)



(b)



(c)



(d)

Fig. 2. Symbolic Trees with Strongest Postconditions or CLP-PROVER (running TRACER on program in Fig. 2(a) with options -intp sp or -intp clp)

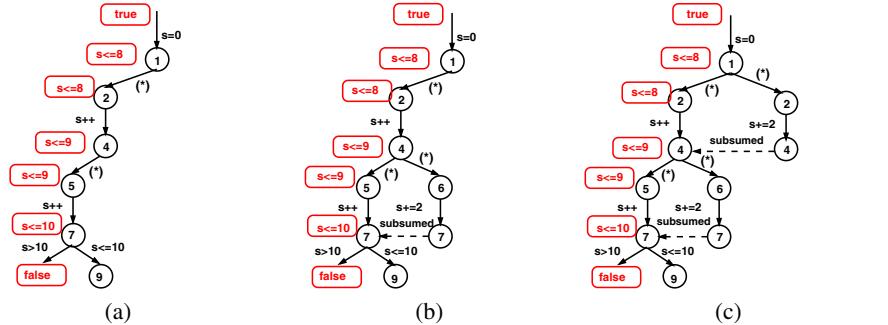


Fig. 3. Symbolic Trees with Weakest Preconditions (running TRACER with -intp wp)

means that the evaluation of the guard can be *true* or *false*. After renaming we obtain the unsatisfiable constraints $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 1 \wedge s_2 > 10$. State-of-the-art interpolation techniques will annotate every location with its corresponding interpolant: $\iota_1 : s_0 \leq 0$, $\iota_2 : s_0 \leq 0$, $\iota_4 : s_1 \leq 1$, $\iota_5 : s_1 \leq 1$, and $\iota_7 : s_2 \leq 2$ where ι_k refers to the interpolant at location k . In all figures, interpolants are enclosed in (red) boxes. Fig. 2(c) shows the tree after the second symbolic path has been explored. At location 7 of the second path TRACER tests if the current symbolic state $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 2$ is subsumed² by $\iota_7 : s_2 \leq 2$, the interpolant at 7. However, this test fails since $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 2 \not\models s_2 \leq 2$. Similarly, TRACER attempts again at location 4 of the third path in Fig. 2(d) if the new symbolic path can be subsumed by a previous explored path. Here, it tests if $s_0 = 0 \wedge s_1 = s_0 + 2$ implies $\iota_4 : s_1 \leq 1$ but again it fails. TRACER can prove the program is safe but the symbolic execution tree built is exponential on the number of program branches. □

² A symbolic state σ is *subsumed* or *covered* by another symbolic state σ' if they refer to same location and the set of states represented by σ is a subset of those represented by σ' . Alternatively, if σ and σ' are seen as formulas then σ is subsumed by σ' if $\sigma \models \sigma'$.

For efficiency, TRACER under-approximates the weakest precondition by a mix of existential quantifier elimination, unsatisfiable cores, and some heuristics. Whenever an infeasible path is detected we compute $\neg(\exists \bar{y} \cdot G)$, the *postcondition* that we want to map into a *precondition*, where G is the guard where the infeasibility is detected and \bar{y} are G -local variables. The two main rules for propagating wp's are:

- (A) $wp(x := e, Q) = Q[e/x]$
- (B) $wp(\text{if}(C) S1 \text{ else } S2, Q) = (C \Rightarrow wp(S1, Q)) \wedge (\neg C \Rightarrow wp(S2, Q))$

Rule (A) replaces all occurrences of x with e in the formula Q . The challenge is how to produce efficient (conjunctive) formulas from rule (B) as weak as possible to increase the likelihood of subsumption. During the forward SE when an infeasible path is detected we discard *irrelevant* guards by using the concept of *unsatisfiable cores* (UC)³ to avoid growing the wp formula unnecessarily. For instance, the formula $C \Rightarrow wp(S1, Q)$ can be replaced with $wp(S1, Q)$ if $C \notin \mathcal{C}$ where \mathcal{C} is a (not necessarily minimal) UC. Otherwise, we underapproximate $C \Rightarrow wp(S1, Q)$ as follows. Let $d_1 \vee \dots \vee d_n$ be $\neg wp(S1, Q)$ then we compute $\bigwedge_{1 \leq i \leq n} (\neg(\exists \bar{x}' \cdot (C \wedge d_i)))$, where existential quantifier elimination removes the post-state variables \bar{x}' . A very effective heuristic if the resulting formula is disjunctive is to delete those conjuncts that are not implied by \mathcal{C} because they are more likely to be irrelevant to the infeasibility reason.

Example 2. Coming back to the program in Fig 2(a). Fig. 3(a) shows the same first symbolic path explored by TRACER but annotated with weakest preconditions: $\iota_1 : s_0 \leq 8$, $\iota_2 : s_0 \leq 8$, $\iota_4 : s_1 \leq 9$, $\iota_5 : s_1 \leq 9$, and $\iota_7 : s_2 \leq 10$. In this example, the wp computations are notably simplified since the guards are clearly irrelevant for the infeasibility of the path, and hence, only rule (A) is triggered. For instance, $\iota_7 : s_2 \leq 10$ is obtained by $\neg(\exists \mathcal{V} \setminus \{s_2\} \cdot s_2 > 10) \equiv s_2 \leq 10$ where \mathcal{V} is the set of all program variables (including renamed variables), and $\iota_6 : s_1 \leq 9$ is obtained by $wp(s_2 = s_1 + 1, s_2 \leq 10) = s_1 \leq 9$. Fig. 3(b) shows the second symbolic path but note that the path can be now subsumed at location 7 since the symbolic state $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 2 \models s_2 \leq 10$. Dashed edges represent subsumed paths and are labelled with "subsumed". Finally, Fig. 3(c) illustrates how the third symbolic path can be also subsumed at location 4 since $s_0 = 0 \wedge s_1 = s_0 + 2 \models s_1 \leq 9$. TRACER proves safety again but the size of the symbolic tree is now linear on the number of branches. \square

With unbounded loops the only hope to produce a proof is *abstraction*. In a nutshell, upon encountering a cycle TRACER computes the *strongest* possible loop invariants $\bar{\Psi}$ by using widening techniques in order to make the SE finite. If a spurious abstract error is found then a *refinement phase* (similar to CEGAR) discovers an interpolant I that rules the spurious error out. After restart, TRACER strengthens $\bar{\Psi}$ by conjoining it with I and the symbolic execution checks *path by path* if the new strengthened formula is loop invariant. If this test fails for a path π , then TRACER unrolls π one more iteration and continues with the process. Notice that the generation of invariants is *dynamic* in the sense that loop unrolls will expose new constraints producing new invariant candidates. For lack of space, we refer readers to [12] for technical details. Here, we illustrate how TRACER handles unbounded loops through the classical example described in Fig 4(a).

³ Given a constraint set S whose conjunction is unsatisfiable, an *unsatisfiable core* (UC) S' is any unsatisfiable subset of S . An UC S' is *minimal* if any strict subset of S' is satisfiable.

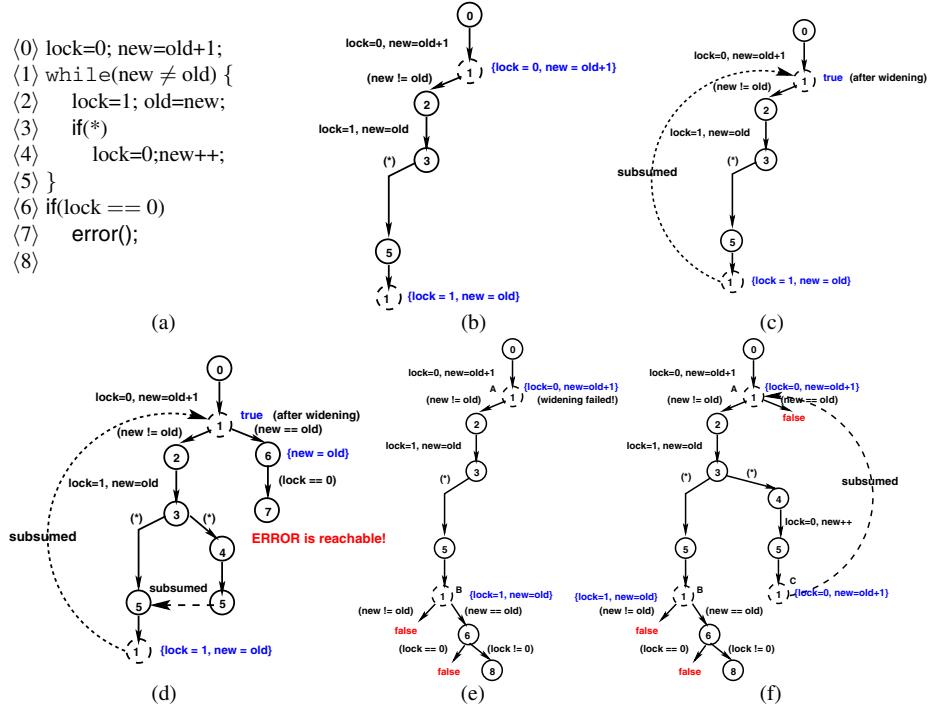


Fig. 4. TRACER execution for an excerpt from a NT Windows driver

Example 3. TRACER executes the program until a cycle is found and checks whether a certain set of loop candidates holds after the execution of the cycle. We obtain the symbolic path $\pi_1 \equiv lock_0 = 0 \wedge new_0 = old_0 + 1 \wedge (new_0 \neq old_0) \wedge lock_1 = 1 \wedge old_1 = new_0$ from executing the `else` branch, shown in Fig. 4(b). Assume a widening ∇ defined as $c \nabla c' \triangleq c$ if $c' \models c$ otherwise $true$, where c and c' are the constraint versions before and after the execution of the cycle corresponding to one candidate. Then, widening our loop candidates (shown between curly brackets in the first occurrence of location 1) $\{lock_0 = 0, new_0 = old_0 + 1\}$ produces an abstracted symbolic state $true ((lock_0 = 0) \nabla (lock_1 = 1)) \equiv true$ and $(new_0 = old_0 + 1) \nabla (old_1 = new_0) \equiv true$. The path π_1 after widening is shown in Fig. 4(c). Note that the symbolic state at the loop header is $true$, and as a result, we can stop executing and avoid unrolling the path π_1 forever since the child (second occurrence of location 1) is subsumed by its parent (first occurrence of 1). We then backtrack to a second path π_2 from executing the `then` branch. For π_2 , the candidates are indeed invariants but this is irrelevant since the execution of π_1 already determined that they were not invariant. As a result of the loss of precision of our abstraction, the exit condition of the loop ($new_0 = old_0$) (Fig. 4(d)) is now satisfied and the error location is reachable by the path $\pi_3 \equiv (new_0 = old_0) \wedge (lock_0 = 0)$. Then, a refinement is triggered. First, we check that π_3 is indeed spurious due to the loop abstraction (i.e., $lock_0 =$

$0 \wedge new_0 = old_0 + 1 \wedge (new_0 = old_0) \wedge (lock_0 = 0)$ is unsatisfiable). Second, by weakest preconditions we infer an interpolant $I \equiv new_0 \neq old_0$ that suffices to rule out the counterexample. Third, we strengthen our loop abstraction *true* with I , record that I cannot be abstracted further, and restart.

After restart, the execution of π_1 shown in Fig. 4(e) cannot be halted at location labelled with B since $(new_0 = old_0 + 1) \nabla (old_1 = new_0)$ is still *true* but this abstraction does not preserve $new_0 \neq old_0$, the interpolant from the refinement phase. As a result, we are not allowed to abstract the candidate $new_0 = old_0 + 1$ at location A and thus the path must be unrolled one more iteration. However, the unrolled path will not take the loop body anymore but follow the exit condition propagating the constraints $lock_1 = 1 \wedge new_1 = old_0$. Hence, the unrolled path is safe. Finally, we explore π_2 from the `then` branch shown in Fig. 4(f). Fortunately, we can stop safely the execution of π_2 (as before) since no abstraction is needed for this path and hence, $new_0 \neq old_0$ is preserved. As a result, the state of the child C is subsumed by its ancestor A . \square

Remarks. It is known that wp may fail to generalize with some loops as Jhala et al. pointed out in [14]. TRACER can be fed with other interpolation methods and/or with inductive invariants from external tools (see Sec. 3). Also, our path invariant technique via widening is closely related to the widening “up to” $S (\nabla^S)$ used in [9], where S contains the constraints inferred by the refinement phase. However, they use it to enhance CEGAR while SE poses different challenges (see [12] Sec.1, Ex.3). Finally, we would like to emphasize that abstraction in TRACER differs from CEGAR in a fundamental way. TRACER attempts at inferring the *strongest* loop invariants modulo the limitations of widening techniques while CEGAR, as well as hybrid tools like CPA-CHECKER and KRATOS, will often propagate coarser abstractions. Although stronger abstractions may be more expensive they may converge faster in presence of loops (see [12] Sec.1, Ex.4).

3 Usage and Implementation

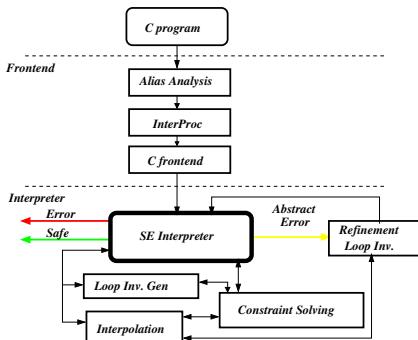


Fig. 5. Implementation of TRACER

Corresponding symbolic execution tree is displayed as the proof object. If the program is unsafe then a counterexample is shown.

Input. TRACER takes as input a C program with assertions of the form `_TRACER_abort(Cond)`, where $Cond$ is a quantifier-free FOL formula. Then, each path that encounters the assertion tests whether $Cond$ holds or not. If yes, the symbolic execution has reached an error node and thus, it reports the error and aborts if the error is real, or refines if spurious. Otherwise, the symbolic execution continues normally.

Output. If the symbolic execution terminates and all `_TRACER_abort` assertions failed then the program is reported as safe and the corre-

Implementation. Fig. 5 outlines the implementation of TRACER. It is divided into two components. First, a C-frontend based on CIL [19] translates the program into a constraint-based logic program. Both pointers and arrays are modeled using the theory of arrays. An alias analysis is used in order to yield sound and finer grained independent partitions (i.e., *separation*) as well as infer which scalars’ addresses may have been taken. Optionally, INTERPROC [17] (option `-loop-inv`) can be used to provide loop invariants. The second component is an interpreter which symbolically executes the constraint-based logic program and it aims at demonstrating that error locations are unreachable. This interpreter is implemented in a *Constraint Logic Programming (CLP)* system called CLP(\mathcal{R}) [11]. Its main sub-components are:

- *Constraint Solving* relies on the CLP(\mathcal{R}) solver to reason fast over linear arithmetic over reals augmented with a decision procedure for arrays (option `-mccarthy`).
- *Interpolation* implements two methods with different logical strength. The first method uses *strongest postconditions* [12][13] (`-intp sp`). The second computes *weakest preconditions* (`-intp wp`) but currently it only supports linear arithmetic over reals. TRACER also provides interfaces to other interpolation methods such as CLP-PROVER (`-intp clp`).
- *Loop Invariant Refinement*. Similar to CEGAR the effectiveness of the refinement phase usually relies on heuristics (`-h` option). But unlike CEGAR tools, SE only performs abstractions at loop headers. Thus, given a path that reaches an error location TRACER only needs to visit those abstraction points in the path and check if one of the them caused the reachability of the error. If yes, it uses interpolation to choose which constraints can rule out the error. Otherwise, the error must be real.
- *Loop Invariant Generation*. If a loop header is found TRACER records a set of *loop invariant* candidates by projecting onto the propagated symbolic state. When a cycle π is found it widens the state at the header by $c\nabla c'$ where c' is the candidate c after the execution of π . Current implementation of widening is $c\nabla c' \triangleq c$ if $c' \models c$ otherwise *true*. Very importantly, if ∇ attempts at abstracting a constraint needed to exclude an error then it fails and the path is unrolled at least one more iteration. Although our experiments show that our method for discovering loop invariants is fast and effective, it is *incomplete* (in the sense that it may cause non-termination) for several reasons. First, the generation of candidates considers only constraints propagated by SE although TRACER allows enriching this set with inductive invariants provided by INTERPROC. Second, the implementation of ∇ is fairly naive. Third, ∇ is applied to each candidate *individually*. By applying ∇ to *all candidate subsets* we could produce richer invariants, although this process would be exponential.

4 Experience with Benchmarks

We ran TRACER on the ntdrivers-simplified and ssh-simplified benchmarks from SV-COMP (sv-comp.sosy-lab.org) and compare with two state-of-the-art tools: CPA-CHECKER [3] and HSF [21]. Fig. 6 shows the results of this comparison including the impact on TRACER using strongest postconditions (SP) and weakest preconditions (WP) as interpolants. Columns 2 and 3 compare the number of states of the symbolic execution tree (#S) explored by TRACER using SP and WP, and columns 4 and 5 compare the number of loop invariant refinements made (#R) using SP and WP. The rest

Program	#S		#R		T			
	SP	WP	SP	WP	SP	WP	CPA	HSF
cdaudio	4663	2138	0	0	12	10	3	529
diskperf	4565	2829	0	0	14	11	3	513
floppy	1758	1357	0	0	4	4	2	568
kfiltr	319	230	0	0	2	2	2	5
s3_clnt_1	∞	6940	∞	33	∞	61	7	8
s3_clnt_2	∞	9871	∞	74	∞	115	12	5
s3_clnt_3	∞	17617	∞	114	∞	338	8	9
s3_clnt_4	∞	6990	∞	46	∞	80	5	8
s3_srvr_1	∞	5496	∞	12	∞	33	18	5
s3_srvr_2	∞	7295	∞	29	∞	120	98	11
s3_srvr_3	∞	5950	∞	14	∞	37	13	39
s3_srvr_4	47988	4349	143	12	372	27	25	10

Fig. 6. Comparison between TRACER and state-of-the-art verifiers on Intel 2.33Ghz 3.2GB

of the columns show total time in seconds T (including compilation time) of TRACER (SP and WP), CPA-CHECKER (CPA), and HSF (HSF). For a fair comparison, TRACER did not use invariants from INTERPROC. ∞ indicates TRACER did not finish within 900 seconds.

Our results indicate that the use of WP pays off with greater gains in programs where TRACER refines heavily, mainly because loop unrolls are expensive for SE, and hence subsuming more often is vital. For ssh-simplified benchmarks (s3_clnt and s3_srvr) TRACER, with SP, was unable to finish for all but one program, where #S, #R and T were about 10-15 times more compared to WP. Compared with HSF, a “pure” CEGAR verifier, TRACER out-performed it in the ntdrivers-simplified benchmarks (first 4 rows) and was out-performed in the rest. This suggests that CEGAR may behave better when numerous loop unrolls are needed and SE may be more suitable when most of the infeasible paths affect safety (where CEGAR would perform many refinements). Comparing with CPA, a hybrid verifier and winner of SV-COMP’12, TRACER fares almost equally in the ntdrivers-simplified benchmarks and s3_srvr programs, but is out-performed in the s3_clnt benchmarks. Nevertheless, our evaluation demonstrates that TRACER is competitive with state-of-the-art verifiers.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
2. Ball, T., Podelski, A., Rajamani, S.K.: Relative Completeness of Abstraction Refinement for Software Model Checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 158–172. Springer, Heidelberg (2002)
3. Beyer, D., et al.: Software Model Checking via Large-Block Encoding. In: FMCAD 2009 (2009)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: BLAST. Int. J. STTT (2007)
5. Cimatti, A., Griggio, A., Micheli, A., Narasamdy, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)

6. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008)
7. Clarke, E., Krong, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
8. Craig, W.: Three Uses of Herbrand-Gentzen Theorem in Relating Model and Proof Theory. JSC (1955)
9. Gulavani, B.S., et al.: Refining Abstract Interpretations. Inf. Process. Lett. (2010)
10. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-SOFT: Software Verification Platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
11. Jaffar, J., Michaylov, S., Stuckey, P., Yap, R.: The CLP(\mathcal{R}) System. TOPLAS (1992)
12. Jaffar, J., Navas, J.A., Santosa, A.E.: Unbounded Symbolic Execution for Program Verification. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 396–411. Springer, Heidelberg (2012); ISBN: 978-3-642-29859-2
13. Jaffar, J., Santosa, A.E., Voicu, R.: An Interpolation Method for CLP Traversal. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 454–469. Springer, Heidelberg (2009)
14. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
15. King, J.: Symbolic Execution and Program Testing. Com. ACM (1976)
16. McMillan, K.L.: An Interpolating Theorem Prover. TCS (2005)
17. Lalire, G., Argoud, M., Jeannet, B.: The Interproc Analyzer,
[http://pop-art.inrialpes.fr/people/bjeannet/
bjeannet-forge/interproc](http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc)
18. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
19. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: CC 2002. LNCS, vol. 2304, p. 213. Springer, Heidelberg (2002)
20. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The YOGI Project: Software Property Checking via Static Analysis and Testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
21. Grebenshchikov, S., et al.: Synthesizing Software Verifiers from Proof Rules. In: PLDI 2012 (2012)
22. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)

Joogie: Infeasible Code Detection for Java

Stephan Arlt¹ and Martin Schäf²

¹ Albert-Ludwigs-Universität Freiburg

² United Nations University, IIIST, Macau

Abstract. We present Joogie, a tool that detects infeasible code in Java programs. Infeasible code is code that does not occur on feasible control-flow paths and thus has no feasible execution. Infeasible code comprises many errors detected by static analysis in modern IDEs such as guaranteed null-pointer dereference or unreachable code. Unlike existing techniques, Joogie identifies infeasible code by proving that a particular statement cannot occur on a terminating execution using techniques from static verification. Thus, Joogie is able to detect infeasible code which is overlooked by existing tools. Joogie works fully automatically, it does not require user-provided specifications and (almost) never produces false warnings.

1 Introduction

We present Joogie, a static analysis tool to detect infeasible code in Java programs. Infeasible code is code which does not occur on any feasible control-flow paths and hence has no feasible execution. That is, infeasible code is either not forward-reachable or not backward-reachable on a feasible execution. Common examples of infeasible code are unreachable code, or guaranteed null-pointer dereference.

Infeasible code tends to occur in a very early stage of development and should be found at the latest during testing. An intrinsic property of infeasible code is that it has *no* feasible execution. That is, a code fragment can be detected to be infeasible without knowing its full context. Extending its context can only restrict its feasible executions and thus an infeasible code fragment will remain infeasible in any larger context. Hence, infeasible code lends itself to be detected by static analysis: it can be detected for code fragments in isolation using relatively coarse abstractions of the feasible executions, and with a very low rate of false warnings.

Infeasible code can, e.g., be detected using data-flow analysis tools such as Findbugs [8] or the built-in static analysis of Eclipse which, among other things, also detects infeasible code. We claim that, among all static analysis tools, those detecting infeasible code are some of the most widely used. Programmers do not suppress Eclipse-warning that an object is always null when dereferenced or that a particular code fragment is unreachable. That is, improving infeasible code detection can have a large impact in practice.

In contrast to existing tools that detect infeasible code, Joogie uses techniques from static verification to prove the presence of infeasible code. This results in

a higher precision than pure syntactic analysis. Joogie first translates a given program into the Boogie language [10] as described in Section 3. Then, a modified version of the Boogie program verification system [1] is used to prove the presence of infeasible code as described in Sect. 4. We show the ability of Joogie to detect infeasible code which is not found using existing tools by applying our tool to three real world applications in Sect. 5. Joogie works fully automatically, does not require any user interaction, and is able to detect real errors while almost never producing false warnings.

2 Joogie Overview

Figure 1 gives an overview of Joogie. Joogie takes a Java program as input. Joogie splits the task of proving the presence of infeasible code in two steps. In a first step, the Java program is translated into Boogie. During this translation, the type system and memory model are replaced by more abstract concepts which facilitate the use of existing verification techniques. The details of this translation are described in Sect. 3. Note that this translation is neither sound nor complete. That is, some feasible executions might be lost which can result in false warnings, and the translation may add feasible executions which can result in false negatives.

In a second step, Joogie calls a modified version of the Boogie program verifier to prove the presence of infeasible code in the Boogie program. The underlying decision procedure is based on the weakest liberal precondition calculus and uses a sound abstraction of the given Boogie program. Section 4 gives more details on the used algorithms. For each infeasible statement in the Boogie program, Joogie reconstructs the corresponding statement in the Java source code and returns an error message. Joogie works fully automatically. Joogie does not require specification statements, but in general it is possible to further annotate the generated Boogie program to increase the detection rate or check for additional properties.

3 Bytecode Translation

Joogie translates Java to Boogie using the Java optimization framework Soot [11]. Soot translates the Java program into a 3-address intermediate representation of the program’s bytecode, which significantly simplifies the translation to Boogie, as only 15 different kinds of statements have to be considered.

One of the most vital parts of translating an object-oriented language into an intermediate verification language is the used memory model. For a sound

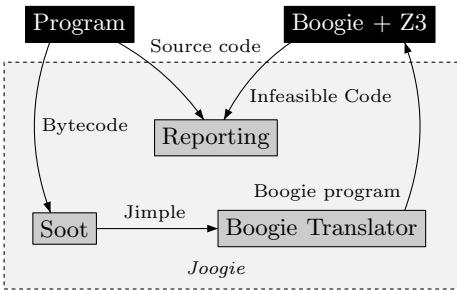


Fig. 1. Overview of Joogie

infeasible code detection it is sufficient to preserve all feasible executions of the original program (contrary to partial correctness proofs, where all infeasible executions have to be preserved). Thus, Joogie can use a simple Burstall-Bornat-style heap-as-array model (see e.g., [4, 10]). The heap is represented by a two-dimensional array, where the first index refers to the address of an object in the heap and the second index refers to the field that is to be accessed. Soot ensures that references to objects are *null* by default. Assertions to guard the heap access are introduced automatically by Joogie. For brevity of exposure, we do not explain this model in detail. Similar approaches can be found, e.g., in Spec# [2] or ESC/Java [5]. Note that using assertions is not sound, as the Java program would throw an exception rather than terminate when the exception is violated.

Integers, Chars, and Bytes are represented using the Boogie built-in type for unbounded natural numbers. Using an unbounded representation for bounded variables is an unsound abstraction. Hence, Joogie uses uninterpreted functions for arithmetic operators, which can be redefined using axioms if a sound handling of primitive types is needed. However, unless the programmer deliberately makes use of Java's overflow and underflow handling, this is a feasible abstraction and, so far, we did not encounter false warnings resulting from this unsoundness. String variables are treated like any other object. Doubles and floats are treated in a similar way as objects. They are represented as arbitrary values and operators on them are represented as uninterpreted functions. This abstraction is coarse and certainly leaves room for improvements, but it is sound and efficient for our purpose of detecting infeasible code. Arrays are represented as one-dimensional unbounded arrays of an appropriate type. The size of an array is stored outside the bounds of the original array. Array-bounds checks are modeled using assertion statements, which is unsound for the general case, as out-of-bounds exceptions might be handled in the code. However, this can be changed easily depending on the user's preferences.

Exceptions are modeled as multiple return parameters of a method. If an exception is thrown, the corresponding return parameter is assigned to the instance of the exception, and the method returns, or, if possible, jumps to an adequate catch block. After each method call, conditional choices are added to redirect the control-flow if an exception has been thrown by the called method.

In general, this translation is not sound as it does not consider aliasing of method parameters and global variables. This unsoundness could be eliminated by, e.g., modeling the aliasing explicitly which would increase the complexity of the translated program significantly. However, our experiments show that this simplification does not introduce false warnings.

4 Infeasible Code Detection

We check for the existence of infeasible code in the Boogie program using the algorithms described in [7] and [3]. These algorithms are implemented as an extension to the Boogie program verification system. For each control location

in a program P , we introduce a statement assigning an auxiliary *reachability variable* r_i to the constant 1, where i ranges over the number of all program statements. This allows us to check the existence of an execution that passes this location, by checking if any terminating execution starting in an initial state where $r_i = 0$ terminates in a state where it is still 0. If this is the case, then no terminating execution passes the assignment $r_i := 1$ and hence no execution passes the considered statement. This check is automated by augmenting the program P with reachability variables, computing a formula representation of the weakest-liberal precondition of this program, and then using a SMT solver (here: Z3) that checks if $(r_i = 0) \models wlp(P, r_i = 0)$ holds (a similar concept is used in [6]).

To compute a formula representation of wlp , we first eliminate the loops in our program P using the abstract loop unwinding from [7]. A loop is replaced by three unwindings. The first and the last unwinding represent the first and the last iteration of the loop, respectively. To every entrance and exit of the middle unwinding, we add non-deterministic assignments to all variables modified inside the loop body. This *abstract unwinding* represents all other unwindings. Note that, for copied locations, we do not create fresh r_i variables, and thus, the abstraction does not remove feasible executions from the program (proof in [7]).

Joogie does not do any inter-procedural analysis. Any procedure call is replaced by a non-deterministic assignment to all variables that might be modified by this procedure. Still, this is a sound abstraction.

For the resulting loop-free program, we compute a formula representation of the weakest-liberal precondition using standard techniques which are already provided by Boogie. The algorithm to detect infeasible code in Boogie programs is sound w.r.t. infeasible code detection under two preconditions: procedure parameters do not alias, and the program is single-threaded. The first one can be lifted by adding switch cases. For multithreading, we do not have a sound solution yet. If a statement is only executed on interleaved executions, it will be reported as infeasible. That is, in general Joogie is not sound. We evaluate its feasibility in the experiments in the next section.

5 Experiments

Joogie, all experimental data, and additional results can be found on the website¹. We apply Joogie on 3 real-world Java applications, TerpWord 4.0, Rachota 2.4, and FreeMind 0.9, to check the performance of Joogie, whether it can find infeasible code, and whether it does produce false warnings. We also apply Joogie on Joogie itself. All experiments are executed several times on a standard notebook (Dual Core 1.6 GHz, 2 GB RAM, 5400 rpm HDD). Note that infeasible code should be detected at the latest during testing, and it should not occur in any stable release of a program. That is, we expect to find hardly any or even no infeasible code. For a detailed evaluation including reports on detection rate, experiments with seeded infeasible code are needed. Table I shows the summary

¹ <http://code.google.com/p/joogie/>

of our experiments, and Figure 2 gives a more detailed view on the computation time per method.

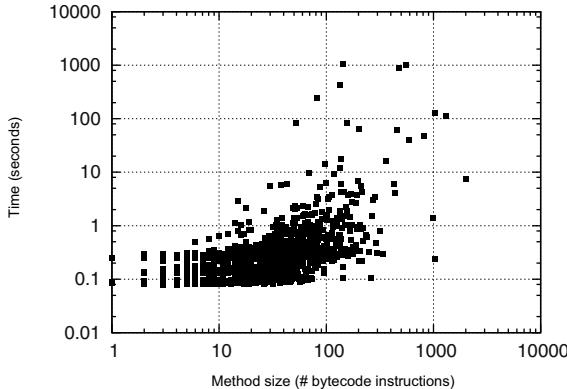


Fig. 2. Computation time of Joogie per Java method

Table 1. Results of applying Joogie to the test applications

Program	LOC	# checked methods	# found bugs	# false warnings	Time (min)
TerpWord	6842	965	4	2	2.95
Rachota	13750	1835	1	0	49.13
FreeMind	40922	8008	12	1	64.41
Joogie	5433	781	0	0	1.37

Observations. Joogie is able to detect infeasible code in the stable releases of 3 applications. Some of it is simple unreachable code, some of it is code that will cause a run time error when reached. Examples of detected infeasible code are given on the Joogie website. We did encounter two false warnings in TerpWord: one is due to a bug when parsing the Java program, the other one is a statement that is only reachable due to interleaving. Joogie does not deal with interleaving. The other sources of unsoundness of the translation from Java to Boogie wrt. infeasible code detection did not cause any false warnings. In Rachota we found one bug. In FreeMind, we found 12 bugs but also 1 false positive due to bugs in Joogie which we could not fix until the deadline.

Figure 2 shows, the average computation time per method is way below one second for most methods. As Joogie is meant to be used incrementally on recently modified program fragments similar to, e.g., the static analysis in Eclipse, the computation time can be tolerated. Larger or more complex methods can be split in smaller parts which are analyzed in isolation.

6 Conclusion

Joogie is useful: it does not require any user interaction, it is fully automatic, it detects errors, and it does almost never produce false warning. The experiments

show that Joogie can be applied to real programs and that it does find infeasible code, even in sufficiently tested code. Our long term goal is to make Joogie efficient enough to run in the background while the programmer is typing. Until then, there is still much room for improvements. The complexity of the generated Boogie program can be further optimized by sharing variables between independent program fragments, techniques from verification could be used to infer invariants, or more efficient ways to represent the heap could be applied.

By using Boogie as an intermediate representation, Joogie can be easily extended by other researchers. E.g., the translation from Java to Boogie could be modified to identify different classes of errors, or specification statements could be added to further increase the detection rate.

We observe that it is not always trivial to understand why code is infeasible. In contrast to, e.g., run-time errors, where a trace counterexample is sufficient to explain why the error occurs, infeasible code can be witnessed by this way. In our future work we will explore techniques like e.g., BugAssist [9] that can be used to explain infeasible control-flow.

Acknowledgements. This work is supported by the projects ARV and COLAB funded by Macau Science and Technology Development Fund.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# Programming System: Challenges and Directions. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)
3. Bertolini, C., Schäf, M., Schweitzer, P.: Infeasible Code Detection. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 310–325. Springer, Heidelberg (2012)
4. Bornat, R.: Proving Pointer Programs in Hoare Logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
5. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. SIGPLAN Not. 37, 234–245 (2002)
6. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011)
7. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: Doomed program points. Form. Methods Syst. Des. 37, 171–199 (2010)
8. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: Companion to OOPSLA 2004, pp. 132–136. ACM, New York (2004)

9. 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)
10. Leino, K.R.M., Rümmer, P.: A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
11. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON 1999, pp. 125–135 (1999)

HECTOR: An Equivalence Checker for a Higher-Order Fragment of ML

David Hopkins¹, Andrzej S. Murawski², and C.-H. Luke Ong¹

¹ Department of Computer Science, University of Oxford, UK

² Department of Computer Science, University of Leicester, UK

Abstract. We present HECTOR, an observational equivalence checker for a higher-order fragment of ML. The input language is RML, the canonical restriction of standard ML to ground-type references. HECTOR accepts programs from a decidable fragment of RML identified by us at ICALP'11, which comprises programs of short-type (order at most 2 and arity at most 1) that may contain free variables whose arguments are also of short-type. This is an expressive fragment that contains complex higher-order types, and includes many examples from the literature which have proven challenging to verify using other methods. To our knowledge, HECTOR is the first fully-automated equivalence checker for higher-order, call-by-value programs. Both sound and complete, the tool relies on the fully abstract game semantics of RML to construct, on-the-fly, visibly pushdown automata which precisely capture program behaviour. These automata are then checked for language equivalence, and if they are inequivalent a counterexample (in the form of a separating context) is constructed.

1 Introduction

ML-like languages combine the power of higher-order functions with imperative constructs and mutable state. We consider the call-by-value language RML, which is essentially the canonical restriction of Standard ML to ground-type references. We are interested in a notion of program equivalence called observational equivalence. Two terms $\Gamma \vdash M_1, M_2$ are *observationally equivalent* just if for every program context $C[-]$ such that $\Gamma \vdash C[M_i] : \text{unit}$, we have that $C[M_1]$ converges if and only if $C[M_2]$ converges. This definition says that two programs are equivalent if one can replace one by the other in any context without affecting the outcome of the computation. Observational equivalence is extremely useful when refactoring or updating code; if the updated version of a function is observationally equivalent to the older version then the changes cannot break any existing code which calls it. This makes observational equivalence an intuitively natural and practically relevant notion of equivalence. Unfortunately, it is notoriously difficult to reason about. Take the programs below.

$$F_1 \equiv \text{let } a = \text{ref } 0 \text{ in let } r = \text{ref } 0 \text{ in } \lambda f. (r := !r + 1; a := f(!r); r := !r - 1; !a)$$

$$F_2 \equiv \lambda f. f(1)$$

It may appear that these two terms should be equivalent, as F_1 uses local variables to return $f(1)$. However, they are separated by the term $G \equiv \lambda F. F(\lambda x. F(\lambda y. y))$. This forces a nested call of F_i . In $G F_1$ this call will be performed before r has been

decremented. Hence, $G F_1$ evaluates to 2 whereas $G F_2$ returns 1. However, the terms $\text{let } c = \text{ref } 0 \text{ in } \lambda f^{\text{unit} \rightarrow \text{unit}}.(c := 0; f(); c := 1; f(); !c)$ and $\lambda f^{\text{unit} \rightarrow \text{unit}}.(f(); f(); 1)$ are equivalent. While the context can use nested calls in the same manner to reset the value of c to 0, any such state changes must be made in a well-bracketed manner and so the terms cannot be separated.

2 Theory and Implementation

We will make use of the fully abstract game semantics of RML [7]. This model views program execution as the playing of a game between the program and its environment. The type sequent $\Gamma \vdash \theta$ determines the rules of a two player game $\llbracket \Gamma \vdash \theta \rrbracket$ to be played between P (the program) and O (the environment). Play proceeds by the players taking it in turn to play a move (which can be either a question or an answer), equipped with a *justification pointer* to an earlier move. These pointers model the variable-to-binder and call-to-return relation within the play. We say a play is *complete* if every question has been answered. The denotation of a program $\Gamma \vdash M : \theta$ is a strategy $\llbracket \Gamma \vdash M \rrbracket$ for playing the game $\llbracket \Gamma \vdash \theta \rrbracket$. Strategies are described using a set of plays which form a playbook telling P how to play. The game model is *fully abstract* in the sense that two programs are observationally equivalent if and only if the sets of complete plays of their denotations are equal [11].

In [7] we identified the *O-strict* fragment of RML. This is the fragment for which the justification pointers from O-moves are always uniquely reconstructible from the underlying move sequence (although those from P-moves can still be ambiguous). This consists of terms-in-context of the shape $x_1 : \Theta_3, \dots, x_n : \Theta_3 \vdash M : \Theta_2$ where Θ_2, Θ_3 are defined as follows.

$$\begin{array}{lll} \Theta_0 ::= \text{unit} \mid \text{int} & \Theta_2 ::= \Theta_0 \mid \Theta_1 \rightarrow \Theta_0 \mid \text{int ref} \\ \Theta_1 ::= \Theta_0 \mid \Theta_0 \rightarrow \Theta_1 \mid \text{int ref} & \Theta_3 ::= \Theta_0 \mid \Theta_2 \rightarrow \Theta_3 \mid \text{int ref} \end{array}$$

If we let a *short* type be a type of order at most two and arity at most one, then the O-strict fragment consists of programs of short types which may contain free identifiers all of whose argument types are short.

We went on to show that the strategies corresponding to terms of the O-strict fragment of RML (with finite data types) can be precisely captured using visibly pushdown automata (VPA). VPA are a subclass of pushdown automata in which the stack action (push, pop, or neither) is determined by the input letter [3]. This gives them highly desirable closure properties; in particular, language equivalence is decidable in polynomial time. Our translation from strategies to VPA allowed us to show that observational equivalence for O-strict terms is EXPTIME-complete [7].

We have now implemented our algorithm into a tool called HECTOR (Higher-order Equivalence Checker for Terms of O-strict RML). Our VPA are constructed inductively over the normal forms of the language, following [7]. Given two such VPA, using a product construction [3], it is easy to construct another to accept their symmetric difference. Then our two programs are equivalent if, and only if, the language accepted by the resulting automaton is empty. We choose to follow an on-the-fly model checking approach as this has proved successful for the game semantics based model checker

MAGE [4]. That is, when constructing our automata, we just return a function from states to the list of transitions out of that state. This function will build up the transition relation only as it is called during our exploration of the automaton. This can allow us to avoid constructing the entire automaton as we can halt the search as soon as a counter-example is found. On-the-fly reachability for pushdown systems using summary edges was described by Alur et al. [2] and we follow their approach. This essentially proceeds as a depth-first search, recording push- and pop-sites so that additional summary edges can be added when two matching transitions are found.

A web interface for HECTOR can be found at <http://mjolnir.cs.ox.ac.uk/~davh/cgi-bin/rml/input/>. Our tool allows programs to be compared, can generate separating contexts where appropriate, and can display the VPA translation of a given term, which represents its game semantics.

3 Examples and Experiments

In this section we consider a number of examples that HECTOR can handle. Where applicable we also compare its performance against HOMER, a game semantics based equivalence checker [8] for the 3rd-order fragment of Idealized Algol (IA). The main difference between RML and IA is that IA uses call-by-name evaluation (and block-allocated variables), which lead to game models that differ significantly [1]. A direct comparison between the two tools is therefore tricky, but we can attempt to use examples which have similar behaviour under both call-by-name and call-by-value evaluation. A further difference is that HOMER does not take advantage of on-the-fly construction but always builds up the entire model.

“Tricky” Examples Several examples in the literature are known to be challenging to verify. In addition to the first inequivalence in Section I due to Stark [12], the following have been analysed respectively by Pitts and Stark [1], and by Dreyer et al. [6].

- (i) $\text{let } c = \text{ref } 0 \text{ in } \lambda f^{\text{unit} \rightarrow \text{unit}}. (c := 1; f(); !c) \cong \lambda f^{\text{unit} \rightarrow \text{unit}}. (f(); 1)$
- (ii) $\text{let } c = \text{ref } 0 \text{ in } \lambda f^{\text{unit} \rightarrow \text{unit}}. (c := 0; f(); c := 1; f(); !c) \cong \lambda f^{\text{unit} \rightarrow \text{unit}}. (f(); f(); 1)$

They are known to be extremely tricky to prove using methods based on logical relations. All three of these examples are in the O-strict fragment and HECTOR can easily handle them as seen in the table below.

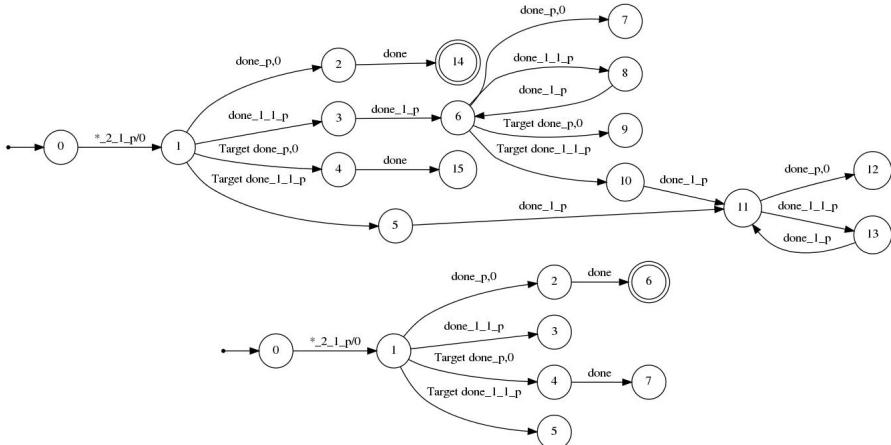
Example	Time to Compare	Time to Generate Counter-Example	State Space
(i) [1]	180ms	N.A.	67
(ii) [6]	130ms	N.A.	231
Sec. 1 [12]	150ms	50ms	57

No-Snapback Another non-obvious example is below.

$$\begin{aligned} p : (\text{unit} \rightarrow \text{unit}) \rightarrow \text{unit} \vdash \\ \text{let } x = \text{ref } 0 \text{ in } p(\lambda y. x := 1); \text{if } !x = 1 \text{ then } \Omega \text{ else } () \cong p(\lambda y. \Omega) \end{aligned}$$

Here Ω is the term which immediately diverges. In the first term, if p ever applies its functional argument to anything then x will be assigned the value 1. This ensures

that when p terminates, the computation will diverge. Conversely, if p does not use its argument then x will have the value 0 so when p finishes the computation terminates. The effect is the same as passing p an argument which will diverge if used. The fact that they are equivalent shows that there is no term which can undo the side-effects caused by running its argument. The VPA translations of these programs as produced by HECTOR are shown below. The reachable states are somewhat different in each case as the divergence occurs at different points. However, in both cases a final state can only be reached if p 's argument is never called.



Scope Extrusion Consider the following terms.

$$M_1 \equiv F : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \vdash$$

let $x = \text{ref } 0$ in $F(\lambda y. \text{if } !x = 0 \text{ then } x := y \text{ else } x := y - 1; !x)$

$$M_2 \equiv F : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \vdash$$

$F(\lambda y. \text{let } x = \text{ref } 0 \text{ in if } !x = 0 \text{ then } x := y \text{ else } x := y - 1; !x)$

$$M_3 \equiv F : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \vdash F(\lambda y. y)$$

The only difference between the first two terms is the location of the $\text{let } x = \text{ref } 0$ in - binding. However, this makes a big difference to their behaviour. In the first term, the value of x persists between calls so when F calls its argument a second time the value in x will be the value of y from the first call. On the other hand, in M_2 a new reference of value 0 is allocated each time the argument is called. Hence, the guard will always be true and so we have $M_1 \not\cong M_2 \cong M_3$. For the inequivalence, HECTOR generates a separating context as a counter-example. A readable version of the context produced is shown on the right. It can be seen that this binds F to a function which applies its argument twice, the first time passing it 1 and the second time 0. It then checks whether the return value from the second call is 0. When M_2 is placed in the context the check will pass as F 's argument is the identity function. However, when M_1 is used the check fails and so the terms are separated.

```
(fun f .
let _ = f (fun g .
  let _ = (g 1)
  in let z = (g 0)
     in assert((z = 0));
        3)
in ())
(fun F .[1])
```

All the examples in this section can be checked by HECTOR in less than a second.

Sorting Sorting algorithms are a challenging example for any model checker due to the complex interplay between control-flow and state. We can use HECTOR to compare different sorting algorithms for equivalence. The table below compares the length of time required to check the equivalence of bubble sort and insert sort on lists of length n containing 3-valued elements. For comparison we include the time taken by HOMER, as well as the state space of the final automaton and the biggest intermediate automaton HOMER produces. As can be seen, HECTOR is outperformed by HOMER. We suspect that this is due to the added complexities of the call-by-value semantics over the call-by-name. However, we can also check the sorting algorithms when they are parameterised by a comparison function $compare : \text{int} \rightarrow \text{int} \rightarrow \text{int}$. In this case a malicious context could pass in a comparison function which does not act as a total order and can use this function to gain more information about the internals of the algorithm. Hence, the two programs are no longer equivalent. Due to the added size of the model when parameterised in this manner, HOMER runs out of memory for lists of length 10. On the other hand, due to the on-the-fly approach HECTOR finds the counter-example almost immediately and so does not have to construct the entire model.

n	HECTOR to Compare	Counter-example	States	HOMER	Final States	Max States
5	3s	N/A	716	1.5s	496	496
7	1min	N/A	5,000	10s	2,800	33,000
10	95min	N/A	120,000	7.5min	60,000	900,000
With A Comparison Function						
5	220ms	120ms	96	2.25min	75,000	75,000
7	225ms	225ms	132	Time Out	Time Out	Time Out
10	300ms	500ms	186	Time Out	Time Out	Time Out
15	400ms	2s	276	Time Out	Time Out	Time Out

Kierstead Terms An interesting family of higher-order terms are the Kierstead terms.

$$K_{n,i} \equiv f : ((\text{unit} \rightarrow \text{unit}) \rightarrow \text{unit}) \rightarrow \text{unit} \vdash f(\lambda x_1.f(\lambda x_2. \dots f(\lambda x_n.x_i()) \dots))$$

For $i \neq j$, $K_{n,i} \not\cong K_{n,j}$. In differentiating these terms the location of justification pointers from P-moves is critical (HECTOR uses tags on the moves to encode the location of these pointers.) We can compare the performance of HECTOR against that of HOMER on the equivalent call-by-name family of Kierstead terms. Again since this is an inequivalence, HECTOR outperforms HOMER as we do not have to construct the entire model. The timing data is shown in the table below.

n	HECTOR to Compare	Counter-example	States	HOMER	Final States	Max States
10	120ms	80ms	150	1s	74	1,400
25	140ms	200ms	366	6s	194	4,000
50	180ms	800ms	576	22s	356	7,000
100	530ms	4.5s	1,600	2min	800	18,000
200	2min	9s	37,000	7min	1,300	42,000

4 Related Work, Conclusions and Further Directions

We have presented HECTOR, an equivalence checker for a higher-order fragment of ML. Our algorithm utilises the fully abstract game semantics of RML. We believe this is the only known procedure for deciding observational equivalence of higher-order ML programs. As HECTOR is the first implementation of this algorithm, a fair comparison with existing tools is difficult. Compared with the call-by-name equivalence checker HOMER, our tool performs much better on inequivalences, thanks to the on-the-fly approach, but not as well on equivalences (which is not surprising as call-by-value game models are more complex constructions [7]). The only other game semantics based verification tool that uses on-the-fly model generation is MAGE [4], which is restricted to 2nd-order, (call-by-name) Idealized Algol programs. MAGE can only check reachability. Other tools, notably TRECS [10] and HMC [9], can verify safety properties of ML programs, but not equivalence.

In future work we hope to expand the language accepted by HECTOR. We know that observational equivalence is undecidable for most types outside the O-strict fragment but there are still a few remaining types whose decidability is unknown. It is also possible to introduce a limited form of recursion into the language, although VPA are no longer sufficiently expressive and we would require the power of DPDA. Additionally, we would like to improve the performance of HECTOR, possibly using predicate abstraction in the style of [5].

References

1. Abramsky, S., McCusker, G.: Call-by-Value Games. In: Nielsen, M. (ed.) CSL 1997. LNCS, vol. 1414, pp. 1–17. Springer, Heidelberg (1998)
2. Alur, R., Chaudhuri, S., Etessami, K., Madhusudan, P.: On-the-Fly Reachability and Cycle Detection for Recursive State Machines. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 61–76. Springer, Heidelberg (2005)
3. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC (2004)
4. Bakewell, A., Ghica, D.R.: On-the-Fly Techniques for Game-Based Software Model Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 78–92. Springer, Heidelberg (2008)
5. Bakewell, A., Ghica, D.R.: Compositional Predicate Abstraction from Game Semantics. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 62–76. Springer, Heidelberg (2009)
6. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. In: ICFP, pp. 143–156 (2010)
7. Hopkins, D., Murawski, A.S., Ong, C.-H.L.: A Fragment of ML Decidable by Visibly Pushdown Automata. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 149–161. Springer, Heidelberg (2011)
8. Hopkins, D., Ong, C.-H.L.: HOMER: A Higher-Order Observational Equivalence Model checkER. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 654–660. Springer, Heidelberg (2009)
9. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: Verifying Functional Programs Using Abstract Interpreters. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 470–485. Springer, Heidelberg (2011)

10. Kobayashi, N.: Model-checking higher-order functions. In: PPDP, pp. 25–36 (2009)
11. Pitts, A.M., Stark, I.D.B.: Operational reasoning for functions with local state. Higher Order Operational Techniques in Semantics (1998)
12. Stark, I.D.B.: Names and Higher-Order Functions. PhD thesis, Univ. of Cambridge (1995)

Resource Aware ML

Jan Hoffmann¹, Klaus Aehlig², and Martin Hofmann²

¹ Yale University

² Ludwig-Maximilians-Universität München

Abstract. The automatic determination of the quantitative resource consumption of programs is a classic research topic which has many applications in software development. Recently, we developed a novel multivariate amortized resource analysis that automatically computes polynomial resource bounds for first-order functional programs.

In this tool paper, we describe Resource Aware ML (RAML), a functional programming language that implements our analysis. Other than in earlier articles, we focus on the practical aspects of the implementation. We describe the syntax of RAML, the code transformation prior to the analysis, the web interface, the output of the analysis, and the results of our experiments with the analysis of example programs.

Keywords: Functional Programming, Static Analysis, Resource Consumption, Quantitative Analysis, Amortized Analysis.

1 Introduction

A quantitative analysis of a program determines the amount of resources, such as memory and time, that the program consumes during its evaluation. Quantitative analyses are needed to compare different algorithms for the same task, to design efficient programs, and to identify performance bottlenecks in software.

Sometimes, it is sufficient to determine the *asymptotic* resource behavior of a program. However, many applications in embedded systems, hard real-time systems, and cloud computing require *concrete* (non-asymptotic) upper bounds for specific hardware. The manual determination of such bounds is not only cumbersome and time consuming but also prone to errors, especially if the analysis has to be repeated after an iteration of the development cycle. As a result, mechanical assistance for the determination of resource bounds is an important and active area of research.

Classic methods for obtaining bounds on the number of loop iterations and recursive calls are based on automatically extracting and solving recurrence relations [1][2][3]. However, both, extracting and solving recurrence relations is a difficult problem. As a result, alternative techniques for the inference of resource bounds have been studied recently. Gulwani et al. propose counter instrumentation and abstract-interpretation-based invariant generation to obtain bounds on loop iterations and function calls [4]. To obtain loop bounds from disjunctive invariants one can use size-change abstraction [5] or proof rules that employ

SMT-solvers [6]. Type-based techniques for automatically inferring bounds on recursive functions are based on sized types [78] or amortized analysis [910], and often restricted to *linear* bounds.

We have recently developed the first type-based resource analysis system that automatically computes *polynomial* resource bounds [111213]. It is inspired by automatic amortized analysis for linear bounds [9]. In a nutshell, we annotate function types with a priori unknown, non-negative rational numbers that represent coefficients of *multivariate resource polynomials*, a class of functions that generalizes non-negative linear combinations of binomial coefficients.¹ A syntax-directed static type analysis then derives linear inequalities for the unknown rational coefficients. Finally, a solution of the resulting linear program with an off-the-shelf LP solver yields a resource polynomial that bounds the resource consumption of the corresponding function. Such an automatic amortized analysis is favorable in the presence of (nested or intermediate) data structures and function composition. See [13] for a detailed comparison with related approaches.

We implemented our multivariate amortized resource analysis in Resource Aware ML (RAML), a first-order, functional language with an ML-like syntax. While one can formalize algorithms and functional programs directly in RAML, it can also be used as a target of resource-preserving translations from other programming languages. In particular we have experimented with a translation from C using the Frama-C framework².

In this tool paper, we describe the current state of development of RAML from a user’s point of view. For a description of the analysis technique that we implemented, please refer to our previous papers [11121314]. Note that the prototype implementation has been used in a previous paper for an experimental evaluation [13]. However, we have never demonstrated the tool at a conference.

2 The Prototype Implementation

The prototype implementation of RAML is written in Haskell and consists of a parser (546 lines of code), a standard type checker (490 lines of code), an interpreter (333 lines of code), an LP solver interface (301 lines of code), and the multivariate analysis system [13] (1637 lines of code). Overall, we needed 4.5 man-months for the implementation of the analysis.

The implementation is well documented and publicly available. The source code of the latest RAML version can be downloaded on the web site of the project [15]. Additionally, there is a web form that can be used to evaluate RAML programs and to compute resource bounds directly on the web.

Extended Syntax. The RAML syntax in the prototype extends the syntax described in our previous papers [1113]. For example, expressions are not restricted to let normal form. We also have more built-in operators and allow a destructive

¹ The user has to provide a maximal degree of the polynomials to limit the number of unknown coefficients.

² <http://frama-c.com>

pattern matching $matchD$ that deallocates the memory cell associated with the matched node of the data structure.

Data types τ are binary trees ($T(\tau)$), lists ($L(\tau)$), integers, Booleans, units, and tuples as defined by the following grammar.

$$\tau ::= \text{int} \mid \text{bool} \mid \text{unit} \mid (\tau_1, \dots, \tau_n) \mid L(\tau) \mid T(\tau)$$

The following EBNF grammar defines expressions e . The reserved function $tick$ is used in the *tick metric* which is described later. The argument q of $tick$ denotes a floating point literal. The operations $binop$ and $unop$ are the usual standard operations for integers and Booleans.

$$\begin{aligned} e ::= & () \mid \text{True} \mid \text{False} \mid n \mid x \mid tick(q) \mid e_1 \text{ binop } e_2 \mid unop \ e \mid f(e_1, \dots, e_n) \\ & \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{if } e \text{ then } e_t \text{ else } e_f \mid [] \mid [e_1, \dots, e_n] \mid (e_1, \dots, e_n) \\ & \mid \text{match } e_1 \text{ with } (x_1, \dots, x_n) \rightarrow e_2 \mid \text{let } (x_1, \dots, x_n) = e_1 \text{ in } e_2 \\ & \mid \text{nil} \mid \text{cons}(e_h, e_t) \mid (\text{match} \mid \text{matchD}) \ e \text{ with } \mid \text{nil} \rightarrow e_1 \mid \text{cons}(x_h, x_t) \rightarrow e_2 \\ & \mid \text{leaf} \mid \text{node}(e_0, e_1, e_2) \mid (\text{match} \mid \text{matchD}) \ e \text{ with } \mid \text{leaf} \rightarrow e_1 \mid \text{node}(x_0, x_1, x_2) \rightarrow e_2 \end{aligned}$$

A RAML program consists of a (possibly empty) list of declarations followed by a main expression. A declaration is either a type declaration $f : \tau_1 \rightarrow \tau_2$ or a function definition $f(x_1, \dots, x_n) = e$, where f is a function name, τ_i are data types, x_i are variables, and e is an expression. There must be exactly one type declaration for every function definition. For every identifier, at most one type declaration and at most one function definition is allowed. Note that one has to provide a monomorphic type for every function in a program. The reason why we avoid polymorphic functions is that the resource consumption of a function depends on its type. Alternatively, we could allow polymorphic functions and analyze a function for each concrete type it is used with in the program.

Destructive Pattern Match. A destructive pattern match—written using $matchD$ —can be used to deallocate memory cells. For instance, in the evaluation of the expression $matchD \ x \text{ with } \mid \text{nil} \rightarrow e_1 \mid \text{cons}(x, xs) \rightarrow e_2$ the memory cell that is referenced in the variable x is deallocated. If memory cells are allocated during the evaluation of e_2 then the deallocated cell may be used to store a new value. So if a deallocated value is accessed during the evaluation of an expression then the behavior of the program is undefined. If used carefully, destructive pattern matches can help to develop and analyze programs that use memory very efficiently. A typical example is an in-place quick-sort algorithm which destructs the input list [15].

Transformation to Let Normal Form. To simplify the resource analysis, we transform the unrestricted RAML expressions of the prototype implementation into expressions in let normal form as defined in [13]. An expression is in let normal form if, whenever possible, term formers are applied to variables only. Furthermore, we make sharing of variables explicit to enable the use of a syntax-directed type rule for sharing of potential in the type inference.

The transformation to let normal from uses a special form of a let expression—called *freelet*—that does not consume any resources. For every expression that

occurs in a position where only variables are allowed, we introduce a new variable with a *freelet*. For technical reasons we also introduce a new variable if the expression in such a *variable only position* in the source program is a variable itself. In this way, it becomes easy to preserve the resource cost of the source program because we know that all variables in the *variable only positions* have been introduced by a *freelet*.

To make sharing explicit, we add an additional syntactic construct to the expression each time a variable occurs multiple times. If a free variable x occurs twice in an expression e , we replace the first occurrence of x with x_1 and the second occurrence of x with x_2 , obtaining a new expression e' . We then replace e with $\text{share}(x, x_1, x_2)$ in e' . In this way, the sharing rule becomes a conventional syntax directed rule in the type inference.

Resource Metrics. Our analysis is parametric in the resource and can deal with every quantity whose consumption in an atomic evaluation step is bounded by a constant. We included three resource metrics in the prototype and it is easy to define more by instantiating the resource constants for the evaluation steps.

The first included metric is the evaluation-step metric that counts the number of evaluation steps in the big-step operational semantics described in [13].

The second metric we included is the heap-space metric. The heap-space used by a node of a data structure depends on the type of the elements of the data structure. That is why we allow the resource constants to depend on the types of the respective expressions in the prototype. For instance, we do not simply have K^{cons} which defines the resource usage of a *cons* but rather $K^{\text{cons}}(A)$ where A is the type of the elements of the list. We define

$$\text{size}(A) = \begin{cases} n & \text{if } A = (A_1, \dots, A_n) \\ 1 & \text{otherwise} \end{cases}$$

Then $K^{\text{cons}}(A) = \text{size}(A) + 1$ is the number of memory cells that are used to store a node of a list of type $L(A)$. Similarly, $K_1^{\text{matCD}}(A) = \text{size}(A) + 1$ memory cells become available in a destructive pattern match. Since the types $L(A)$ are known at compile time, it makes no difference for the analysis whether the constants depend on data types. In principle, the values of these constants could depend on anything that is statically known about the program. However, the current implementation limits this dependency to type information.

The third implemented metric measures the number of ticks that occur in an evaluation. To this end, a programmer can insert expressions such as $\text{tick}(3.5)$ or $\text{tick}(-4)$ into the code. Every time the expression $\text{tick}(q)$ is evaluated, q resources are consumed, or $-q$ resources become available if q is negative. The tick metric can be used to manually model specific resource metrics and is helpful for testing.

A table with the values of the constants in the metrics can be found in [14].

Web Interface. The source code of the prototype is available for download on the RAML website [15]. Alternatively, programs can be executed directly on the web with input in a text field or selection of example files from a drop-down menu. A second text field contains the output of the RAML prototype.

One can use the web interface to compute resource bounds for a program or to evaluate the main expression. The following options are available.

1. The resource metric to be used in the analysis. It can either be heap-space consumption, evaluation steps or ticks.
2. An upper bound on the maximal degree that can occur in the resource bounds. If the degree is too low then the analysis reports that the linear program is infeasible.
3. Whether to have verbose output. The verbose output shows for instance the function definitions in let normal form.

Output of the Analysis. The result of a successful evaluation is the value of the main expression as well as the number of heap cells, the number of evaluation steps, and the number of ticks that have been used during the evaluation.

The output of a resource analysis is either a list of symbolic bounds along with refined typing information—one for each function in the program including the main expression—or an error message. If the program is type correct then the only error that can occur is the message *the linear program is infeasible*. It indicates that the LP solver finished unsuccessfully and that RAML was thus not able to compute a bound for the program. This often, but not necessarily, implies that the resource usage of the given program cannot be bounded by a polynomial of the given degree.

Of course, as with any static analysis, there also exist polynomially bounded programs for which RAML cannot compute bounds. For instance, the analysis often fails if recursion is guarded by a Boolean function as opposed to the constructors of a data structure. This is often the case in programs whose resource consumption depends on the values of integers. Nevertheless, the analysis works well for recursive functions that use inductive data types and pattern matching.

Below is the output of the analysis with the evaluation-step metric for the function *quicksort* in the file *quicksort.raml* which can be found online [15].

```
> raml analyse eval-steps 3 quicksort.raml
quicksort: L(int) -> L(int)
Positive annotations of the argument      Positive annotations of the result
0 -> 3.0      1 -> 26.0      2 -> 24.0
```

The number of evaluation steps consumed by quicksort is at most:
 $12.0*n^2 + 14.0*n + 3.0$
where n is the length of the input

It contains the type of the function and the potential annotations of the argument type and the result type. Finally, the potential annotations are converted into a usual polynomial for the convenience of the user. This transformation is a combination of a change of basis from binomial coefficients to the common basis and the abstraction from sizes of individual inner data structures to their maximal size. The exact meaning of the type annotations is described in our earlier work [13]. Note, however, that they may carry more detailed information than the symbolic bound. Also note that only non-zero annotations are shown in the output and that the resource annotations of the output type are all zero.

3 Experiments

We successfully applied the analysis to a wide range of examples from functional programming such as sorting algorithms, matrix multiplication, breadth-first search, and longest common subsequence via dynamic programming.

In most cases, the derived evaluation-step and heap-space bounds were asymptotically tight. The analysis works efficiently and only needs a few seconds, even on larger programs. We also compared our computed bounds with the measured worst-case resource consumption of the programs and found that the constants factors are often close or even identical to the optimal ones.

The analyzed programs, tables with running times and computed bounds, and plots that show the bounds and the measured costs are available online [15] and in the first author's dissertation [14].

References

1. Wegbreit, B.: Mechanical Program Analysis. *Commun. ACM* 18(9), 528–539 (1975)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 161–203 (2011)
4. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL 2009), pp. 127–139 (2009)
5. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound Analysis of Imperative Programs with the Size-Change Abstraction. In: Yahav, E. (ed.) *SAS 2011*. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011)
6. Gulwani, S., Zuleger, F.: The Reachability-Bound Problem. In: Conf. on Prog. Lang. Design and Impl. (PLDI 2010), pp. 292–304 (2010)
7. Chin, W.-N., Khoo, S.-C.: Calculating Sized Types. *High.-Ord. and Symb. Comp.* 14(2-3), 261–300 (2001)
8. Vasconcelos, P.: Space Cost Analysis Using Sized Types. PhD thesis, School of Computer Science, University of St Andrews (2008)
9. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL 2003), pp. 185–197 (2003)
10. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: Sestoft, P. (ed.) *ESOP 2006*. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
11. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: Gordon, A.D. (ed.) *ESOP 2010*. LNCS, vol. 6012, pp. 287–306. Springer, Heidelberg (2010)
12. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: Ueda, K. (ed.) *APLAS 2010*. LNCS, vol. 6461, pp. 172–187. Springer, Heidelberg (2010)
13. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th Symp. on Principles of Prog. Langs. (POPL 2011) (2011)
14. Hoffmann, J.: Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis. PhD thesis, Ludwig-Maximilians-Universität München (2011)
15. Hoffmann, J., et al.: RAML Web Site, <http://raml.tcs.ifi.lmu.de>

Author Index

- Aehlig, Klaus 781
Aiken, Alex 71, 394
Albarghouthi, Aws 672
Alberti, Francesco 679
Alglaive, Jade 495
Alur, Rajeev 495
Arlt, Stephan 767
Armstrong, Philip 699
Arun-Kumar, S. 444
Atig, Mohamed Faouzi 210
August, David 751

Bakst, Alexander 744
Benque, David 686
Berdine, Josh 155
Bodik, Rastislav 3
Bogomolov, Sergiy 479
Bohy, Aaron 652
Bouajjani, Ahmed 210
Bourton, Sam 686
Bradley, Aaron R. 4, 532
Brázdil, Tomáš 23
Brockschmidt, Marc 105
Bruttomesso, Roberto 679
Bruyère, Véronique 652
Buckl, Christian 658

Chandra, Satish 599
Chatterjee, Krishnendu 23
Chaudhuri, Swarat 732
Chechik, Marsha 672
Chen, Yu-Fang 55
Cheng, Chih-Hong 658
Chu, Duc-Hiep 616
Cimatti, Alessandro 277, 378
Clarke, Edmund M. 310
Cockerton, Caitlin 686
Conchon, Sylvain 718
Cook, Byron 686
Corvino, Raffaele 378
Cox, Arlen 155

De Paula, Flavio M. 513
Derrick, John 243
Dill, David L. 2

Dillig, Isil 394
Dillig, Thomas 394
Dong, Jin Song 705
Driscoll, Evan 665

Ehlers, Rüdiger 39
Emmi, Michael 210
Esparza, Javier 7, 123

Filiot, Emmanuel 652
Fisher, Jasmin 686
Fredrikson, Matthew 548
Frehse, Goran 479

Gaiser, Andreas 123
Geisinger, Michael 658
Ghilardi, Silvio 679
Giesl, Jürgen 105
Goel, Amit 718
Goldsmith, Michael 699
Griggio, Alberto 277
Grosu, Radu 479
Guet, Călin C. 294
Guha, Shibashis 444
Gulwani, Sumit 634
Gupta, Ashutosh 294
Gurfinkel, Arie 672

Hague, Matthew 260
Han, Cheng-Shen 410
Harris, William R. 581
Hassan, Zyad 532
Hasuo, Ichiro 462
Hawblitzel, Chris 712
Henzinger, Thomas A. 294
Hoffmann, Jan 781
Hofmann, Martin 781
Hopkins, David 774
Hu, Alan J. 513

Ishtiaq, Samin 155, 686

Jaffar, Joxan 616, 758
Jegourel, Cyrille 327
Jha, Somesh 548, 581
Jhala, Ranjit 744

- Jiang, Jie-Hong Roland 410
 Jin, Naiyong 652
 Joiner, Richard 548
- Kahlon, Vineet 227
 Kawaguchi, Ming 712, 744
 Kiefer, Stefan 123, 693
 Knoll, Alois 658
 Komuravelli, Anvesh 310
 Köpf, Boris 564
 Křetínský, Jan 7
 Krstić, Sava 718
 Kučera, Antonín 23
- Ladan, Hamed 479
 Lahiri, Shuvendu K. 427, 712
 Lal, Akash 210, 427
 Lazzaro, Armando 378
 Lee, Wonchan 88
 Legay, Axel 327
 Li, Yi 672
 Lin, Anthony Widjaja 260
 Liu, Feng 751
 Liu, Yang 705
 Lowe, Gavin 699
- Mador-Haim, Sela 495
 Majumdar, Rupak 362
 Malik, Sharad 751
 Maranget, Luc 495
 Martin, Milo M.K. 495
 Mateescu, Maria 294
 Mauborgne, Laurent 564
 McMillan, Kenneth L. 394
 Mebsout, Alain 718
 Memarian, Kayvan 495
 Moskal, Michał 6
 Murali, Vijayaraghavan 758
 Murawski, Andrzej S. 693, 774
 Musiol, Richard 105
 Myers, Chris J. 5
- Nahir, Amir 513
 Narasamdy, Iman 378
 Narayan, Chinmay 444
 Navas, Jorge A. 758
 Nori, Aditya V. 71
 Novotný, Petr 23
- Ochoa, Martín 564
 Ong, C.-H. Luke 774
- Otto, Carsten 105
 Ouaknine, Joël 693, 699
 Owens, Scott 495
- Palikareva, Hristina 699
 Păsăreanu, Corina S. 310
 Piterman, Nir 686
 Podelski, Andreas 479
 Porras, Phillip 548
- Qadeer, Shaz 427
- Ranise, Silvio 679
 Raskin, Jean-François 652
 Rebêlo, Henrique 712
 Reps, Thomas 174, 548, 581, 665
 Rizzo, Tiziana 378
 Rollini, Simone Fulvio 193
 Rondon, Patrick 744
 Roscoe, A.W. 699
 Roveri, Marco 378
 Ruess, Harald 658
- Saïdi, Hassen 548
 Sankaranarayanan, Sriram 343
 Sanseviero, Angela 378
 Santosa, Andrew E. 758
 Sarkar, Susmit 495
 Schäf, Martin 767
 Schellhorn, Gerhard 243
 Schwartz-Narbonne, Daniel 751
 Sedwards, Sean 327
 Sery, Ondrej 193
 Sewell, Peter 495
 Sezgin, Ali 294
 Sharma, Rahul 71
 Sharygina, Natasha 193, 679
 Singh, Rishabh 634, 738
 Singhania, Nimit 599
 Sinha, Nishant 599
 Solar-Lezama, Armando 732, 738
 Somenzi, Fabio 532
 Song, Songzheng 705
 Sridharan, Manu 599
 Suenaga, Kohei 462
 Sun, Jun 705
- Taylor, Alex 686
 Tchaltsev, Andrei 378
 Thakur, Aditya 174, 665

- Thomas, Wolfgang 1
Tiwari, Ashish 343, 725
Torlak, Emin 3

Vardi, Moshe Y. 686
Venet, Arnaud J. 139

Wachter, Björn 693
Wang, Bow-Yaw 55, 88
Wang, Chao 227
Wehrheim, Heike 243

Wehrle, Martin 479
Williams, Derek 495
Wintersteiger, Christoph M. 155
Worrell, James 693, 699

Yegneswaran, Vinod 548
Yi, Kwangkeun 88

Zaïdi, Fatiha 718
Zamani, Majid 362
Zutshi, Aditya 343