ARCoSS

LNCS 7795

Nir Piterman
Scott A. Smolka (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

19th International Conference, TACAS 2013
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013
Rome, Italy, March 2013, Proceedings

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Springer

Nir Piterman   Scott A. Smolka (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

19th International Conference, TACAS 2013
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2013
Rome, Italy, March 16-24, 2013
Proceedings

Springer

Volume Editors

Nir Piterman
University of Leicester
Department of Computer Science
University Road, LE1 7RH Leicester, UK
E-mail: nir.piterman@le.ac.uk

Scott A. Smolka
Stony Brook University
Department of Computer Science
Stony Brook, NY 11794-4400, USA
E-mail: sas@cs.sunysb.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36741-0 e-ISBN 978-3-642-36742-7
DOI 10.1007/978-3-642-36742-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013932470

CR Subject Classification (1998): F.3.1-2, D.2.4-5, F.2.2, D.3.1, D.3.3, D.1.1, F.1.1-3, C.3

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

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)

Foreword

ETAPS 2013 is the sixteenth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised six sister conferences (CC, ESOP, FASE, FOSSACS, POST, TACAS), 20 satellite workshops (ACCAT, AiSOS, BX, BYTECODE, CerCo, DICE, FESCA, GRAPHITE, GT-VMT, HAS, Hot-Spot, FSS, MBT, MEALS, MLQA, PLACES, QAPL, SR, TERMGRAPH and VSSE), three invited tutorials (*e-education*, by John Mitchell; *cyber-physical systems*, by Martin Fränzle; and *e-voting* by Rolf Küsters) and eight invited lectures (excluding those specific to the satellite events).

The six main conferences received this year 627 submissions (including 18 tool demonstration papers), 153 of which were accepted (6 tool demos), giving an overall acceptance rate just above 24%. (ETAPS 2013 also received 11 submissions to the software competition, and 10 of them resulted in short papers in the TACAS proceedings). Congratulations therefore to all the authors who made it to the final programme! I hope that most of the other authors will still have found a way to participate in this exciting event, and that you will all continue to submit to ETAPS and contribute to making it the best conference on software science and engineering.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis, security and improvement. The languages, methodologies and tools that support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a confederation in which each event retains its own identity, with a separate Programme Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronised parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for 'unifying' talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2013 was organised by the *Department of Computer Science of 'Sapienza' University of Rome*, in cooperation with

- ▷ European Association for Theoretical Computer Science (EATCS)
- ▷ European Association for Programming Languages and Systems (EAPLS)
- ▷ European Association of Software Science and Technology (EASST).

The organising team comprised:

General Chair:	*Daniele Gorla;*
Conferences:	*Francesco Parisi Presicce;*
Satellite Events:	*Paolo Bottoni* and *Pietro Cenciarelli;*
Web Master:	*Igor Melatti;*
Publicity:	*Ivano Salvo;*
Treasurers:	*Federico Mari* and *Enrico Tronci.*

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Vladimiro Sassone (Southampton, chair), Martín Abadi (Santa Cruz), Erika Ábrahám (Aachen), Roberto Amadio (Paris 7), Gilles Barthe (IMDEA-Software), David Basin (Zürich), Saddek Bensalem (Grenoble), Michael O'Boyle (Edinburgh), Giuseppe Castagna (CNRS Paris), Albert Cohen (Paris), Vittorio Cortellessa (L'Aquila), Koen De Bosschere (Gent), Ranjit Jhala (San Diego), Matthias Felleisen (Boston), Philippa Gardner (Imperial College London), Stefania Gnesi (Pisa), Andrew D. Gordon (MSR Cambridge and Edinburgh), Daniele Gorla (Rome), Klaus Havelund (JLP NASA Pasadena), Reiko Heckel (Leicester), Holger Hermanns (Saarbrücken), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Steve Kremer (Nancy), Gerald Lüttgen (Bamberg), Tiziana Margaria (Potsdam), Fabio Martinelli (Pisa), John Mitchell (Stanford), Anca Muscholl (Bordeaux), Catuscia Palamidessi (INRIA Paris), Frank Pfenning (Pittsburgh), Nir Piterman (Leicester), Arend Rensink (Twente), Don Sannella (Edinburgh), Zhong Shao (Yale), Scott A. Smolka (Stony Brook), Gabriele Taentzer (Marburg), Tarmo Uustalu (Tallinn), Dániel Varró (Budapest) and Lenore Zuck (Chicago).

The ordinary running of ETAPS is handled by its management group comprising: Vladimiro Sassone (chair), Joost-Pieter Katoen (deputy chair and publicity chair), Gerald Lüttgen (treasurer), Giuseppe Castagna (satellite events chair), Holger Hermanns (liaison with local organiser) and Gilles Barthe (industry liaison).

I would like to express here my sincere gratitude to all the people and organisations that contributed to ETAPS 2013, the Programme Committee chairs and members of the ETAPS conferences, the organisers of the satellite events, the speakers themselves, the many reviewers, all the participants, and Springer-Verlag for agreeing to publish the ETAPS proceedings in the ARCoSS subline.

Last but not least, I would like to thank the organising chair of ETAPS 2013, Daniele Gorla, and his Organising Committee, for arranging for us to have ETAPS in the most beautiful and historic city of Rome.

My thoughts today are with two special people, profoundly different for style and personality, yet profoundly similar for the love and dedication to our discipline, for the way they shaped their respective research fields, and for the admiration and respect that their work commands. Both are role-model computer scientists for us all.

ETAPS in Rome celebrates *Corrado Böhm*. Corrado turns 90 this year, and we are just so lucky to have the chance to celebrate the event in Rome, where he has worked since 1974 and established a world-renowned school of computer scientists. Corrado has been a pioneer in research on programming languages and their semantics. Back in 1951, years before FORTRAN and LISP, he defined and implemented a *metacircular compiler* for a programming language of his invention. The compiler consisted of just 114 instructions, and anticipated some modern list-processing techniques.

Yet, Corrado's claim to fame is asserted through the breakthroughs expressed by the *Böhm-Jacopini Theorem* (CACM 1966) and by the invention of *Böhm-trees*. The former states that any algorithm can be implemented using only sequencing, conditionals, and while-loops over elementary instructions. Böhm trees arose as a convenient data structure in Corrado's milestone proof of the decidability inside the λ-calculus of the equivalence of terms in β-η-normal form.

Throughout his career, Corrado showed exceptional commitment to his roles of researcher and educator, fascinating his students with his creativity, passion and curiosity in research. Everybody who has worked with him or studied under his supervision agrees that he combines an outstanding technical ability and originality of thought with great personal charm, sweetness and kindness. This is an unusual combination in problem-solvers of such a high calibre, yet another reason why we are ecstatic to celebrate him. *Happy birthday from ETAPS, Corrado!*

ETAPS in Rome also celebrates the life and work of *Kohei Honda*. Kohei passed away suddenly and prematurely on December 4th, 2012, leaving the saddest gap in our community. He was a dedicated, passionate, enthusiastic scientist and –more than that!– his enthusiasm was contagious. Kohei was one of the few theoreticians I met who really succeeded in building bridges between theoreticians and practitioners. He worked with W3C on the standardisation of web services choreography description languages (WS-CDL) and with several companies on *Savara* and *Scribble*, his own language for the description of application-level protocols among communicating systems.

Among Kohei's milestone research, I would like to mention his 1991 epoch-making paper at ECOOP (with M. Tokoro) on the treatment of asynchrony in message passing calculi, which has influenced all process calculi research since. At ETAPS 1998 he introduced (with V. Vasconcelos and M. Kubo) a new concept in type theories for communicating processes: it came to be known as '*session types*,' and has since spawned an entire research area, with practical and multi-disciplinary applications that Kohei was just starting to explore.

Kohei leaves behind him enormous impact, and a lasting legacy. He is irreplaceable, and I for one am proud to have been his colleague and glad for the opportunity to arrange for his commemoration at ETAPS 2013.

My final ETAPS '*Foreword*' seems like a good place for a short reflection on ETAPS, what it has achieved in the past few years, and what the future might have in store for it.

On April 1st, 2011 in Saarbrücken, we took a significant step towards the consolidation of ETAPS: the establishment of *ETAPS e.V.* This is a *non-profit association* founded under German law with the immediate purpose of supporting the conference and the related activities. ETAPS e.V. was required for practical reasons, e.g., the conference needed (to be represented by) a legal body to better support authors, organisers and attendees by, e.g., signing contracts with service providers such as publishers and professional meeting organisers. Our ambition is however to make of '*ETAPS the association*' more than just the organisers of '*ETAPS the conference*'. We are working towards finding a voice and developing a range of activities to support our scientific community, in cooperation with the relevant existing associations, learned societies and interest groups. The process of defining the structure, scope and strategy of ETAPS e.V. is underway, as is its first ever membership campaign. For the time being, ETAPS e.V. has started to support community-driven initiatives such as open access publications (LMCS and EPTCS) and conference management systems (Easychair), and to cooperate with cognate associations (European Forum for ICT).

After two successful runs, we continue to support POST, *Principles of Security and Trust*, as a candidate to become a permanent ETAPS conference. POST was the first addition to our main programme since 1998, when the original five conferences met together in Lisbon for the first ETAPS. POST resulted from several smaller workshops and informal gatherings, supported by IFIP WG 1.7, and combines the practically important subject of security and trust with strong technical connections to traditional ETAPS areas. POST is now attracting interest and support from prominent scientists who have accepted to serve as PC chairs, invited speakers and tutorialists. I am very happy about the decision we made to create and promote POST, and to invite it to be a part of ETAPS.

Considerable attention was recently devoted to our *internal processes* in order to streamline our procedures for appointing Programme Committees, choosing invited speakers, awarding prizes and selecting papers; to strengthen each member conference's own Steering Group, and, at the same time, to strike a balance between these and the ETAPS Steering Committee. A lot was done and a lot remains to be done.

We produced a *handbook* for local organisers and one for PC chairs. The latter sets out a code of conduct that all the people involved in the selection of papers, from PC chairs to referees, are expected to adhere to. From the point of view of the authors, we adopted a *two-phase submission* protocol, with fixed deadlines in the first week of October. We published a *confidentiality policy* to set high standards for the handling of submissions, and a *republication policy* to clarify what kind of material remains eligible for submission to ETAPS after presentation at a workshop. We started an *author rebuttal phase*, adopted by most of the conferences, to improve the author experience. It is important to acknowledge that – regardless of our best intentions and efforts – the quality of reviews is not always what we would like it to be. To remain true to our commitment to the authors who elect to submit to ETAPS, we must endeavour to improve our standards of refereeing. The rebuttal phase is a step in that direction and, according to our experience, it seems to work remarkably well at little cost, provided both authors and PC members use it for what it is. ETAPS has now reached a healthy paper acceptance rate around the 25% mark, essentially uniformly across the six conferences. This seems to me to strike an excellent balance between being selective and being inclusive, and I hope it will be possible to maintain it even if the number of submissions increases.

ETAPS signed a favourable three-year publication contract with Springer for publication in the ARCoSS subline of LNCS. This was the result of lengthy negotiations, and I consider it a good achievement for ETAPS. Yet, publication of its proceedings is possibly the hardest challenge that ETAPS – and indeed most computing conferences – currently face. I was invited to represent ETAPS at a most interesting Dagstuhl Perspective Workshop on the '*Publication Culture in Computing Research*' (seminar 12452). The paper I gave there is available online from the workshop proceedings, and illustrates three of the views I formed also thanks to my experience as chair of ETAPS, respectively on open access, bibliometrics, and the roles and relative merits of conferences versus journal publications. Open access is a key issue for a conference like ETAPS. Yet, in my view it does not follow that we can altogether dispense with publishers – be they commercial, academic, or learned societies – and with their costs. A promising way forward may be based on the '*author-pays*' model, where publications fees are kept low by resorting to learned-societies as publishers. Also, I believe it is ultimately in the interest of our community to de-emphasise the perceived value of conference publications as viable – if not altogether superior – alternatives to journals. A large and ambitious conference like ETAPS ought to be able to rely on quality open-access journals to cover its entire spectrum of interests, even if that means promoting the creation of a new journal.

Due to its size and the complexity of its programme, hosting ETAPS is an increasingly challenging task. Even though excellent candidate *locations* keep being volunteered, in the longer run it seems advisable for ETAPS to provide more support to local organisers, starting e.g., by taking direct control of the organisation of satellite events. Also, after sixteen splendid years, this may be a good time to start thinking about exporting ETAPS to other continents. The US East Coast would appear to be the obvious destination for a first ETAPS outside Europe.

The strength and success of ETAPS comes also from presenting – regardless of the natural internal differences – a homogeneous interface to authors and participants, i.e., to look like one large, coherent, well-integrated conference rather than a mere co-location of events. I therefore feel it is vital for ETAPS to regulate the centrifugal forces that arise naturally in a ‘union’ like ours, as well as the legitimate aspiration of individual PC chairs to run things their way. In this respect, we have large and solid foundations, alongside a few relevant issues on which ETAPS has not yet found agreement. They include, e.g., submission by PC members, rotation of PC memberships, and the adoption of a rebuttal phase. More work is required on these and similar matters.

January 2013

Vladimiro Sassone
ETAPS SC Chair
ETAPS e.V. President

Preface

This volume contains the proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2013 took place during March 18–21, 2013, in the eternal city of Rome, Italy.It was part of the 16th European Joint Conference on Theory and Practice of Software (ETAPS 2013).

TACAS is a forum for researchers, developers, and users interested in rigorously based tools and algorithms for the construction and analysis of systems. The research areas covered by TACAS include, but are not limited to, formal methods, software and hardware verification, static analysis, programming languages, software engineering, real-time systems, communication protocols, and biological systems. TACAS provides a venue where common problems, heuristics, algorithms, data structures, and methodologies in these areas can be discussed and explored.

Following a debut in 2012, TACAS 2013 solicited four kinds of papers, including three types of full-length papers (15 pages), as well as short tool demonstration papers (6 pages):

- Research papers – papers describing novel research on topics included in the remit of TACAS.
- Case study papers – papers reporting on case studies (preferably in a "real life" setting), describing methodologies and approaches used.
- Regular tool papers – papers describing a tool (either completely new, new component, or existing) and focusing on engineering aspects of the tool (including, e.g., software architecture, data structures, and algorithms).
- Tool demonstration papers – papers focusing on the usage aspects of tools relevant to the above-mentioned topics.

This year, TACAS attracted a total of 172 paper submissions, divided into 130 research papers, 15 regular tool papers, 9 case study papers, and 18 tool demonstration papers. Each submission was refereed by at least three reviewers, who evaluated the papers, commented on them, and in many cases suggested improvements and enhancements. The reviewing process was followed by an online Program Committee discussion. As a result of the discussion, 42 papers were accepted for presentation at the conference: 32 research papers, 1 case study paper, 3 regular tool papers, and 6 tool demonstration papers.

TACAS 2013 marked the second time that the Competition on Software Verification was associated with TACAS. This volume includes an overview of the competition results, and short papers describing 10 of the 11 tools that participated in the competition. These papers were reviewed by a separate Program Committee and each paper was refereed by at least three reviewers. Competition results were presented at the conference by Dirk Beyer, the Competition Chair, and the verifiers were presented by the participating teams.

In addition to refereed contributions, the program included an invited talk by Orna Grumberg. TACAS took place in an exciting and vibrant scientific atmosphere, consisting of five other sister conferences (CC, ESOP, FASE, FoSSaCS, and Post), with (sometimes) overlapping scientific fields of interest, their invited speakers, and the ETAPS unifying speakers Gilles Barthe and Cédric Fournet.

We would like to thank all of the authors who submitted papers to TACAS 2013, the Program Committee members, and additional reviewers, without whom TACAS would not have been such a success. We would especially like to thank Claude Marche for his invaluable help as TACAS Tool Chair. We also benefited greatly from the EasyChair conference management system, which we used to run the Program Commitee discussion and to handle the submission, review, and proceedings preparation process. Finally, we would like to thank the TACAS Steering Committee, the ETAPS Steering Committee, and the ETAPS Organizing Committee chaired by Daniele Gorla, who made ETAPS 2013 such a memorable event.

January 2013

Nir Piterman
Scott Smolka

Organization

Steering Committee

Rance Cleaveland	University of Maryland, USA
Kim Guldstrand Larsen	Aalborg University, Denmark
Joost-Pieter Katoen	RWTH Aachen University, Germany
Bernhard Steffen	TU Dortmund, Germany
Lenore Zuck	University of Illinois in Chicago, USA

Program Committee

Erika Abraham	RWTH Aachen University, Germany
Marsha Chechik	University of Toronto, Canada
Rance Cleaveland	University of Maryland, USA
Leonardo De Moura	Microsoft Research, USA
Cindy Eisner	IBM Research - Haifa, Israel
Cédric Fournet	Microsoft, UK
Dimitra Giannakopoulou	NASA Ames, USA
Susanne Graf	VERIMAG (CNRS and Grenoble University), France
Kim Guldstrand Larsen	Aalborg University, Denmark
Klaus Havelund	NASA JPL, USA
Laurie Hendren	McGill University, Canada
Gerard Holzmann	NASA JPL, USA
Michael Huth	Imperial College London, UK
Paola Inverardi	Università dell'Aquila, Italy
Joost-Pieter Katoen	RWTH Aachen University, Germany
Panagiotis Katsaros	Aristotle University of Thessaloniki, Greece
Hillel Kugler	Microsoft Research, UK
Barbara König	Universität Duisburg-Essen, Germany
Axel Legay	IRISA/INRIA, Rennes, France
Claude Marche	INRIA, France
Tobias Nipkow	TU Munich, Germany
Gethin Norman	University of Glasgow, UK
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Nir Piterman	University of Leicester, UK
Mooly Sagiv	Tel Aviv University, Israel
Natasha Sharygina	University of Lugano, Switzerland
Scott Smolka	State University of New York, USA

Bernhard Steffen	TU Dortmund, Germany
Cesare Tinelli	The University of Iowa, USA
Verena Wolf	Saarland University, Germany
Lenore Zuck	University of Illinois in Chicago, USA

Program Committee for Competition on Software Verification

Dirk Beyer	University of Passau, Germany
Lucas Cordeiro	Universidade Federal do Amazonas, Brazil
Bernd Fischer	University of Southampton, UK
Arie Gurfinkel	SEI, USA
Matthias Heizmann	University of Freiburg, Germany
Stefan Löwe	University of Passau, Germany
Vadim Mutilin	ISP RAS, Russian Federation
Andrey Rybalchenko	TU Munich, Germany
Carsten Sinz	Karlsruhe Institute of Technology, Germany
Jiri Slaby	Masaryk University, Czech Republic
Tomáš Vojnar	Brno University of Technology, Czech Republic
Philipp Wendler	University of Passau, Germany

Additional Reviewers

Abdeddaïm, Yasmina
Albarghouthi, Aws
Alberti, Francesco
Aleksandrowicz, Gadi
Andreychenko, Aleksandr
Autili, Marco
Bacci, Giovanni
Balasubramanian, Daniel
Banach, Richard
Barringer, Howard
Bartocci, Ezio
Ben-David, Shoham
Bensalem, Saddek
Bhaduri, Purandar
Bjørner, Nikolaj
Blanchette, Jasmin Christian
Blume, Christoph
Bobot, François
Bollig, Benedikt
Bolton, Matthew
Bonakdarpour, Borzoo
Bouajjani, Ahmed
Boyer, Benoît
Bozga, Marius
Bozzelli, Laura
Bruggink, Harrie Jan Sander
Brumley, David
Bøgholm, Thomas
Chatzieleftheriou, George
Chen, Xin
Chen, Yu-Fang
Chockler, Hana
Copty, Fady
Corzilius, Florian
D'Silva, Vijay
de Boer, Pieter-Tjerk
Dehnert, Christian
Delahaye, Benoit
Deshpande, Tushar
Deters, Morgan
Eades Iii, Harley
Efraimidis, Pavlos
Fahrenberg, Uli
Fedyukovich, Grigory
Feo, Sergio
Fisman, Dana
Florian, Mihai
Fontana, Peter
Fränzle, Martin
Fu, Hongfei
Ganesh, Vijay

Garrison, Bill
Grechanik, Mark
Gretz, Friedrich
Griggio, Alberto
Gurfinkel, Arie
Haller, Leopold
Han, Tingting
Hansen, Henri
Harkjær Møller, Mikael
Heinen, Jonathan
Heljanko, Keijo
Howar, Falk
Hyvärinen, Antti
Hölzl, Johannes
Iosif, Radu
Isberner, Malte
Ivrii, Alexander
Jaeger, Manfred
Jansen, Christina
Jansen, Nils
Jegourel, Cyrille
Jovanovic, Dejan
Jung, Yungbum
Kerstan, Henning
Kidd, Nicholas
Kiefer, Stefan
Komuravelli, Anvesh
Krüger, Thilo
Kupferman, Orna
Küpper, Sebastian
Laarman, Alfons
Lamprecht, Anna-Lena
Langerak, Rom
Lee, Adam
Loup, Ulrich
Luckow, Kasper
Manevich, Roman
Mardare, Radu
Margalit, Oded
Markey, Nicolas
Merten, Maik
Michael, Maged
Mikeev, Linar
Mikucionis, Marius
Miller, Alice
Mogavero, Fabio
Monniaux, David
Moy, Matthieu
Musial, Peter
Møller, Anders
Namjoshi, Kedar
Naujokat, Stefan
Nestmann, Uwe
Neubauer, Johannes
Neuhäußer, Martin R.
Nevo, Ziv
Nickovi, Dejan
Noll, Thomas
Oe, Duckki
Olesen, Mads C.
Pai, Ganesh
Parker, David
Pelliccione, Patrizio
Person, Suzette
Potet, Marie-Laure
Poulsen, Danny Bøgsted
Pradella, Matteo
Pulungan, Reza
Quinton, Sophie
Rajan, Hridesh
Ranzato, Francesco
Ravn, Anders
Raymond, Pascal
Reger, Giles
Reineke, Jan
Reynolds, Andrew
Rinetzky, Noam
Rojas, José Miguel
Rollini, Simone Fulvio
Rozier, Kristin Yvonne
Ruah, Sitvanit
Rungta, Neha
Rydeheard, David
Rüthing, Oliver
Sadre, Ramin
Saidi, Hassen
Salay, Rick
Samanta, Roopsha
Schmalz, Matthias
Schneider-Kamp, Peter
Schubert, Tobias
Schulze, Christoph
Schwoon, Stefan
Sery, Ondrej
Shacham, Ohad
Shafiei, Nastaran
Sher, Falak
Simon, Axel
Sims, Steve
Sinha, Nishant
Spieler, David
Sproston, Jeremy
Srba, Jiri
Srivathsan, Baluguru
Stachtiari, Emmanouela
Stoelinga, Mariëlle I.A.
Stoller, Scott
Stückrath, Jan
Suter, Philippe
Telek, Miklos
Tivoli, Massimo
Tkachuk, Oksana
Vafeiadis, Viktor
van Breugel, Franck
van de Pol, Jaco
Veksler, Tatyana
Viswanathan, Mahesh
von Styp, Sabrina
Wang, Bow-Yaw
Wimmer, Ralf
Windmüller, Stephan
Yang, Bow-Yaw
Yang, Guowei
Yang, Junxing
Yorav, Karen
Yue, Haidi
Zhang, Lijun
Ziv, Avi
Zuliani, Paolo

Additional Reviewers for Competition on Software Verification

Falke, Stephan	Mandrykin, Mikhail	Zakharov, Ilya

SAT-Based Model Checking: Interpolation, IC3 and beyond (Invited Talk)

Orna Grumberg

Computer Science Department, The Technion, Haifa, Israel

Model checking [3] is an automatic approach to formally verifying that a given system satisfies a given specification. The system to be verified is modelled as a finite state machine and the specification is described using temporal logic. Model checking algorithms are typically based on an exploration of the model state space while searching for violations of the specification. In spite of its great success in verifying hardware and software systems, the applicability of model checking is impeded by its high space and time requirements. This is referred to as the *state explosion problem*.

The introduction of SAT-based model checking algorithms [1, 8, 6, 9, 2] significantly increases the size of the systems that can be model checked. In its early days SAT-based model checking was used mostly for bug hunting. The introduction of *interpolation* enabled an efficient complete algorithm, referred to as Interpolation-based model checking (ITP) [6]. ITP uses interpolation to extract an over-approximation of a set of reachable states from a proof of unsatisfiability generated by a SAT-solver. The set of reachable states computed by the reachability analysis is used by ITP to check if a system M satisfies a safety property AGp.

In [2] an alternative SAT-based algorithm, called IC3, is introduced. Similarly to ITP, IC3 also computes over-approximations of sets of reachable states. However, ITP unrolls the model in order to obtain more precise approximations. In many cases, this is a bottleneck of the approach. IC3, on the other hand, improves the precision of the approximations by performing many local checks that do not require unrolling.

Here, we survey several approaches to enhancing SAT-based model checking. One approach, detailed in [9], uses *interpolation sequence* [5, 7] rather than interpolation in order to obtain a more precise over-approximation of the set of reachable states.

The other approach, described in [10], integrates lazy abstraction with IC3 in order to achieve scalability. *Lazy abstraction* [4, 7], originally developed for software model checking, is a specific type of abstraction that allows hiding different model details at different steps of the verification. We find the IC3 algorithm most suitable for lazy abstraction since its state traversal is performed by means of *local* reachability checks, each involving only two consecutive sets. A different abstraction can therefore be applied in each of the local checks.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS/ETAPS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
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. Clarke, E.C., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
4. Henzinger, T., Jhala, R., Majumdar, R.: Lazy abstraction. In: POPL (2002)
5. Jhala, R., McMillan, K.L.: Interpolant-Based Transition Relation Approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
6. 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)
7. 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)
8. 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)
9. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: FMCAD (2009)
10. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and SAT-based reachability in hardware model checking. In: FMCAD (2012)

Table of Contents

On-the-Fly Exact Computation of Bisimilarity Distances[*]

Giorgio Bacci[1,2], Giovanni Bacci[2], Kim G. Larsen[2], and Radu Mardare[2]

[1] Dept. of Mathematics and Computer Science, University of Udine, Italy
giorgio.bacci@uniud.it
[2] Department of Computer Science, Aalborg University, Denmark
{grbacci,giovbacci,kgl,mardare}@cs.aau.dk

Abstract. This paper proposes an algorithm for exact computation of bisimilarity distances between discrete-time Markov chains introduced by Desharnais et. al. Our work is inspired by the theoretical results presented by Chen et. al. at FoSSaCS'12, proving that these distances can be computed in polynomial time using the ellipsoid method. Despite its theoretical importance, the ellipsoid method is known to be inefficient in practice. To overcome this problem, we propose an efficient on-the-fly algorithm which, unlike other existing solutions, computes exactly the distances between given states and avoids the exhaustive state space exploration. It is parametric in a given set of states for which we want to compute the distances. Our technique successively refines over-approximations of the target distances using a greedy strategy which ensures that the state space is further explored only when the current approximations are improved. Tests performed on a consistent set of (pseudo)randomly generated Markov chains shows that our algorithm improves, on average, the efficiency of the corresponding iterative algorithms with orders of magnitude.

1 Introduction

Probabilistic bisimulation for Markov chains (MCs), introduced by Larsen and Skou [12], is the key concept for reasoning about the equivalence of probabilistic systems. However, when one focuses on quantitative behaviours it becomes obvious that such an equivalence is too "exact" for many purposes as it only relates processes with identical behaviours. In various applications, such as systems biology [15], games [3], planning [6] or security [2], we are interested in knowing whether two processes that may differ by a small amount in the real-valued parameters (probabilities) have "sufficiently" similar behaviours. This motivated the development of the metric theory for MCs, initiated by Desharnais et al. [8] and greatly developed and explored by van Breugel, Worrell and others [17,16]. It consists in proposing a *bisimilarity distance (pseudometric)*, which measures the

[*] Work supported by Sapere Aude: DFF-Young Researchers Grant 10-085054 of the Danish Council for Independent Research, by the VKR Center of Excellence MT-LAB and by the Sino-Danish Basic Research Center IDEA4CPS.

behavioural similarity of two MCs. The pseudometric proposed by Desharnais et al. is parametric in a *discount factor* $\lambda \in (0, 1]$ that controls the significance of the future in the measurement.

Since van Breugel et. al. have presented a fixed point characterization of the aforementioned pseudometric in [16], several iterative algorithms have been developed in order to compute its approximation up to any degree of accuracy [9,17,16]. Recently, Chen et. al. [4] proved that, for finite MCs with rational transition function, the bisimilarity pseudometrics can be computed exactly in polynomial time. The proof consists in describing the pseudometric as the solution of a linear program that can be solved using the *ellipsoid method.* Although the ellipsoid method is theoretically efficient, *"computational experiments with the method are very discouraging and it is in practice by no means a competitor of the, theoretically inefficient, simplex method"*, as stated in [14]. Unfortunately, in this case the simplex method cannot be used to speed up performances in practice, since the linear program to be solved may have an exponential number of constraints in the number of states of the MC.

In this paper, we propose an alternative approach to this problem, which allows us to compute the pseudometric exactly and efficiently in practice. This is inspired by the characterization of the undiscounted pseudometric using *couplings*, given in [4], which we extend to generic discount factors. A coupling for a pair of states of a given MC is a function that describes a possible redistribution of the transition probabilities of the two states; it is evaluated by the *discrepancy function* that measures the behavioural disimilarities between the two states. In [4] it is shown that the bisimilarity pseudometric for a given MC is the minimum among the discrepancy functions corresponding to all the couplings that can be defined for that MC; moreover, the bisimilarity pseudometric is itself a discrepancy function corresponding to an *optimal coupling.* This suggests that the problem of computing the pseudometric can be reduced to the problem of finding a coupling with the least discrepancy function.

Our approach aims at finding an optimal coupling using a greedy strategy that starts from an arbitrary coupling and repeatedly looks for new couplings that improve the discrepancy function. This strategy will eventually find an optimal coupling. We use it to support the design of an on-the-fly algorithm for computing the exact behavioural pseudometric that can be either applied to compute all the distances in the model or to compute only some designated distances. The advantage of using an on-the-fly approach consists in the fact that we do not need to exhaustively explore the state space nor to construct entire couplings but only those fragments that are needed in the local computation.

The efficiency of our algorithm has been evaluated on a significant set of randomly generated MCs. The results show that our algorithm performs orders of magnitude better than the corresponding iterative algorithms proposed, for instance in [9,4]. Moreover, we provide empirical evidence for the fact that our algorithm enjoys good execution running times.

One of the main practical advantages of our approach consists in the fact that it can focus on computing only the distances between states that are of particular interest. This is useful in practice, for instance when large systems are considered and visiting the entire state space is expensive. A similar issue has been considered by Comanici et. al., in [5], who have noticed that for computing the approximated pseudometric one does not need to update the current value for all the pairs at each iteration, but it is sufficient to only focus on the pairs where changes are happening rapidly. Our approach goes much beyond this idea. Firstly, we are not only looking at approximations of the bisimilarity distance, but we develop an exact algorithm; secondly, we provide a termination condition that can be checked locally, still ensuring that the local optimum corresponds to the global one. In addition, our method can be applied to decide whether two states of an MC are probabilistic bisimilar, to identify the bisimilarity classes for a given MC or to solve lumpability problems. Our approach can also be used with approximation techniques as, for instance, to provide a least over-approximation of the behavioural distance given over-estimates of some particular distances. This can be further integrated with other approximate algorithms having the advantage of the on-the-fly state space exploration.

Synopsis: The paper is organized as follows. In Section 2, we recall the basic preliminaries on Markov chains and define the bisimilarity pseudometric, for which we provide an alternative characterization in Section 3. Section 4 collects all theoretical results which are the basis for the development of the on-the-fly algorithm, presented in Section 5, for the exact computation of the pseudometric. The efficiency of our algorithm is supported by experimental results, shown in Section 6. Final remarks and conclusions are in Section 7.

2 Markov Chains and Bisimilarity Pseudometrics

In this section we give the definitions of *(discrete-time) Markov chains* (MCs) and *probabilistic bisimilarity* for MCs [12]. Then we recall the *bisimilarity pseudometric* of Desharnais et. al. [8], but rather than giving its first logical definition, we present its fixed point characterization given by van Breugel et. al. [16].

Definition 1 (Markov chain). *A* (discrete-time) Markov chain *is a tuple* $\mathcal{M} = (S, A, \pi, \ell)$ *consisting of a countable nonempty set* S *of* states, *a nonempty set* A *of* labels, *a* transition probability function $\pi \colon S \times S \to [0, 1]$ *such that, for arbitrary* $s \in S$, $\sum_{t \in S} \pi(s, t) = 1$, *and a* labelling function $\ell \colon S \to A$. $\mathcal{M}$ *is* finite *if its support set* S *is finite.*

Given a finite MC $\mathcal{M} = (S, A, \pi, \ell)$, we identify the transition probability function π with its *transition matrix* $(\pi(s, t))_{s,t \in S}$. For $s, t \in S$, we denote by $\pi(s, \cdot)$ and $\pi(\cdot, t)$, respectively, the probability distribution of exiting from s to any state and the sub-probability distribution of entering to t from any state.

The MC $\mathcal{M}$ induces an *underlying (directed) graph*, denoted by $\mathcal{G}(\mathcal{M})$, where the states act as vertices and (s, t) is an edge in $\mathcal{G}(\mathcal{M})$, if and only if, $\pi(s, t) > 0$. For a subset $Q \subseteq S$, we denote by $R_{\mathcal{M}}(Q)$ the set of states reachable from some $s \in Q$, and by $R_{\mathcal{M}}(s)$ we denote $R_{\mathcal{M}}(\{s\})$.

From a theoretical point of view, it is irrelevant whether the transition probability function of a given Markov Chain has rational values or not. However, for algorithmic purposes, in this paper we assume that for arbitrary $s, t \in S$, $\pi(s,t) \in \mathbb{Q} \cap [0,1]$. For computational reasons, in the rest of the paper we restrict our investigation to finite Markov chains.

Definition 2 (Probabilistic Bisimulation). *Let $\mathcal{M} = (S, A, \pi, \ell)$ be an MC. An equivalence relation $R \subseteq S \times S$ is a* probabilistic bisimulation *if whenever $s \, R \, t$, then*

(i) $\ell(s) = \ell(t)$ and,
(ii) for each R-equivalence class E, $\sum_{u \in E} \pi(s,u) = \sum_{u \in E} \pi(t,u)$.

Two states $s, t \in S$ are bisimilar*, written $s \sim t$, if they are related by some probabilistic bisimulation.*

This definition is due to Larsen and Skou [12]. Intuitively, two states are bisimilar if they have the same label and their probability of moving by a single transition to any given equivalence class is always the same.

The notion of equivalence can be relaxed by means of a pseudometric, which tells us how far apart from each other two elements are and whenever they are at zero distance they are equivalent. The bisimilarity pseudometric of Desharnais et. al. [8] on MCs enjoys the property that two states are at zero distance if and only if they are bisimilar. This pseudometric can be defined as the least fixed point of an operator based on the Kantorovich metric for comparing probability distributions, which makes use of the notion of *matching*.

Definition 3 (Matching). *Let $\mu, \nu \colon S \to [0,1]$ be probability distributions on S. A* matching for the pair (μ, ν) *is a probability distribution $\omega \colon S \times S \to [0,1]$ on $S \times S$ satisfying*

$$\forall\, u \in S.\ \textstyle\sum_{s \in S} \omega(u,s) = \mu(u)\,, \qquad \forall\, v \in S.\ \textstyle\sum_{s \in S} \omega(s,v) = \nu(v)\,. \tag{1}$$

We call μ and ν, respectively, the left and the right marginals *of ω.*

In the following, we denote by $\mu \otimes \nu$ the set of all matchings for (μ, ν).

Remark 4. Note that, for S finite, (1) describes the constraints of a homogeneous transportation problem (TP) [7,10], where the vector $(\mu(u))_{u \in S}$ specifies the amounts to be shipped and $(\nu(v))_{v \in S}$ the amounts to be received. Thus, a matching ω for (μ, ν) induces a matrix $(\omega(u,v))_{u,v \in S}$ to be thought as a shipping schedule belonging to the transportation polytope $\mu \otimes \nu$. Hereafter, we denote by $TP(c, \nu, \mu)$ the TP with cost matrix $(c(u,v))_{u,v \in S}$ and marginals ν and μ. ∎

For $\mathcal{M} = (S, A, \pi, \ell)$ an MC, and $\lambda \in (0,1]$ a *discount factor*, the operator $\Delta_\lambda^{\mathcal{M}} \colon [0,1]^{S \times S} \to [0,1]^{S \times S}$, for $d \colon S \times S \to [0,1]$ and $s, t \in S$, is defined by:

$$\Delta_\lambda^{\mathcal{M}}(d)(s,t) = \begin{cases} 1 & \text{if } \ell(s) \neq \ell(t) \\ \lambda \cdot \displaystyle\min_{\omega \in \pi(s,\cdot) \otimes \pi(t,\cdot)} \sum_{u,v \in S} d(u,v) \cdot \omega(u,v) & \text{if } \ell(s) = \ell(t) \end{cases}$$

In the above definition, $\pi(s,\cdot) \otimes \pi(t,\cdot)$ is a closed polytope so that the minimum is well defined and it corresponds to the optimal value of $TP(d, \pi(s,\cdot), \pi(t,\cdot))$.

The set $[0,1]^{S\times S}$ is endowed with the partial order $\sqsubseteq$ defined as $d \sqsubseteq d'$ iff $d(s,t) \leq d'(s,t)$ for all $s,t \in S$. This forms a complete lattice with bottom element $\mathbf{0}$ and top element $\mathbf{1}$, defined as $\mathbf{0}(s,t) = 0$ and $\mathbf{1}(s,t) = 1$, for all $s,t \in S$. For $D \subseteq [0,1]^{S\times S}$, the least upper bound $\bigsqcup D$, and greatest lower bound $\bigsqcap D$ are given by $(\bigsqcup D)(s,t) = \sup_{d\in D} d(s,t)$ and $(\bigsqcap D)(s,t) = \inf_{d\in D} d(s,t)$ for all $s,t \in S$.

In [16], for any $\mathcal{M}$ and $\lambda \in [0,1]$, $\Delta_\lambda^{\mathcal{M}}$ is proved to be monotonic, thus, by Tarski's fixed point theorem, it admits least and greatest fixed points.

Definition 5 (Bisimilarity pseudometric). *Let $\mathcal{M}$ be an MC and $\lambda \in (0,1]$ be a discount factor, then the λ-discounted* bisimilarity pseudometric *for $\mathcal{M}$, written $\delta_\lambda^{\mathcal{M}}$, is the least fixed point of $\Delta_\lambda^{\mathcal{M}}$.*

Hereafter, $\Delta_\lambda^{\mathcal{M}}$ and $\delta_\lambda^{\mathcal{M}}$ will be denoted simply by Δ_λ and δ_λ, respectively, when the Markov chain $\mathcal{M}$ is clear from the context.

3 Alternative Characterization of the Pseudometric

In [4], Chen et. al. proposed an alternative characterization of δ_1, relating the pseudometric to the notion of *coupling*. In this section, we recall the definition of coupling, and generalize the characterization for generic discount factors.

Definition 6 (Coupling). *Let $\mathcal{M} = (S, A, \pi, \ell)$ be a finite MC. The Markov chain $\mathcal{C} = (S \times S, A \times A, \omega, l)$ is said a* coupling for $\mathcal{M}$ *if, for all $s,t \in S$,*

1. *$\omega((s,t),\cdot) \in \pi(s,\cdot) \otimes \pi(t,\cdot)$, and*
2. *$l(s,t) = (\ell(s), \ell(t))$.*

A coupling for $\mathcal{M}$ can be seen as a probabilistic pairing of two copies of $\mathcal{M}$ running synchronously, although not necessarily independently. Couplings have been used to characterize weak ergodicity of arbitrary Markov chains [11], or to give upper bounds on convergence to stationary distributions [1,13].

Given a coupling $\mathcal{C} = (S \times S, A \times A, \omega, l)$ for $\mathcal{M} = (S, A, \pi, \ell)$ we define $\Gamma_\lambda^{\mathcal{C}} \colon [0,1]^{S\times S} \to [0,1]^{S\times S}$ for $d \colon S \times S \to [0,1]$ and $s,t \in S$, as follows:

$$\Gamma_\lambda^{\mathcal{C}}(d)(s,t) = \begin{cases} 1 & \text{if } \ell(s) \neq \ell(t) \\ \lambda \cdot \displaystyle\sum_{u,v\in S} d(u,v) \cdot \omega((s,t),(u,v)) & \text{if } \ell(s) = \ell(t) \end{cases}$$

One can easily verify that, for any $\lambda \in (0,1]$, $\Gamma_\lambda^{\mathcal{C}}$ is well-defined, moreover it is order preserving. By Tarski's fixed point theorem, $\Gamma_\lambda^{\mathcal{C}}$ admits a least fixed point, which we denote by $\gamma_\lambda^{\mathcal{C}}$. In Section 4.1 we will see that, for any $s,t \in S$, $\gamma_1^{\mathcal{C}}(s,t)$ corresponds to the probability of reaching a state (u,v) with $\ell(u) \neq \ell(v)$ starting from the state (s,t) in the underling graph of $\mathcal{C}$. For this reason we will call $\gamma_\lambda^{\mathcal{C}}$ the λ-*discounted discrepancy* of $\mathcal{C}$ or simply the λ-discrepancy of $\mathcal{C}$.

Lemma 7. *Let $\mathcal{M}$ be an MC, $\mathcal{C}$ be a coupling for $\mathcal{M}$, and $\lambda \in (0,1]$ be a discount factor. If $d = \Gamma_\lambda^{\mathcal{C}}(d)$ then $\delta_\lambda \sqsubseteq d$.*

As a consequence of Lemma 7 we obtain the following characterization for δ_λ, which generalizes [4, Theorem 8] for generic discount factors.

Theorem 8 (Minimum coupling criterion). *Let $\mathcal{M}$ be an MC and $\lambda \in (0,1]$ be a discount factor. Then, $\delta_\lambda = \min\left\{\gamma_\lambda^{\mathcal{C}} \mid \mathcal{C}\text{ coupling for }\mathcal{M}\right\}$.*

Proof. For any fixed $d \in [0,1]^{S\times S}$ there exists a coupling $\mathcal{C}$ for $\mathcal{M}$ such that $\Gamma_\lambda^{\mathcal{C}}(d) = \Delta_\lambda(d)$. Indeed we can take as transition function for $\mathcal{C}$, the joint probability distribution ω such that, for all $s,t \in S$, $\sum_{u,v\in S} d(u,v)\cdot\omega((s,t),(u,v))$ achieves the minimum value.

Let $\mathcal{D}$ be a coupling for $\mathcal{M}$ such that $\Gamma_\lambda^{\mathcal{D}}(\delta_\lambda) = \Delta_\lambda(\delta_\lambda)$. By Definition 5, $\Delta_\lambda(\delta_\lambda) = \delta_\lambda$, therefore δ_λ is a fixed point for $\Gamma_\lambda^{\mathcal{D}}$. By Lemma 7, δ_λ is a lower bound of the set of fixed points of $\Gamma_\lambda^{\mathcal{D}}$, therefore $\delta_\lambda = \gamma_\lambda^{\mathcal{D}}$. By Lemma 7, we have also that, for any coupling $\mathcal{C}$ of $\mathcal{M}$, $\delta_\lambda \sqsubseteq \gamma_\lambda^{\mathcal{C}}$. Therefore, given the set $D = \left\{\gamma_\lambda^{\mathcal{C}} \mid \mathcal{C}\text{ coupling for }\mathcal{M}\right\}$, it follows that $\delta_\lambda \in D$ and δ_λ is a lower bound for D. Hence, by antisymmetry of $\sqsubseteq$, $\delta_\lambda = \min D$. □

4 Exact Computation of Bisimilarity Distance

Inspired by the characterization given in Theorem 8, in this section we propose a procedure to exactly compute the bisimilarity pseudometric.

For $\lambda \in (0,1]$, the set of couplings for $\mathcal{M}$ can be endowed with the preorder $\trianglelefteq_\lambda$ defined as $\mathcal{C} \trianglelefteq_\lambda \mathcal{D}$, if and only if, $\gamma_\lambda^{\mathcal{C}} \sqsubseteq \gamma_\lambda^{\mathcal{D}}$. Theorem 8 suggests to look at all the couplings $\mathcal{C}$ for $\mathcal{M}$ in order to find an optimal one, i.e., minimal w.r.t. $\trianglelefteq_\lambda$. However, it is clear that the enumeration of all the couplings is unfeasible, therefore it is crucial to provide an efficient search strategy which prevents us to do that. Moreover we also need an efficient method for computing the λ-discrepancy.

In Subsection 4.1 the problem of computing the λ-discrepancy of a coupling $\mathcal{C}$ is reduced to the problem of computing reachability probabilities in $\mathcal{C}$. Then, Subsection 4.2 illustrates a greedy strategy that explores the set of couplings until an optimal one is eventually reached.

4.1 Computing the λ-Discrepancy

In this section, we first recall the problem of computing the reachability probability for general MCs [1], then we instantiate it to compute the λ-discrepancy.

Let $\mathcal{M} = (S, A, \pi, \ell)$ be an MC, and x_s denote the probability of reaching $G \subseteq S$ from $s \in S$. The goal is to compute x_s for all $s \in S$. The following holds

$$x_s = 1 \quad \text{if } s \in G\,, \qquad\qquad x_s = \textstyle\sum_{t\in S} x_t \cdot \pi(s,t) \quad \text{if } s \in S \setminus G\,, \tag{2}$$

that is, either G is already reached, or it can be reached by way of another state. Equation (2) defines a linear equation system of the form $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$, where $S_? = S \setminus G$, $\boldsymbol{x} = (x_s)_{s\in S_?}$, $\boldsymbol{A} = (\pi(s,t))_{s,t\in S_?}$, and $\boldsymbol{b} = (\sum_{t\in G}\pi(s,t))_{s\in S_?}$.

This linear equation system always admits a solution in $[0,1]^S$, however, it may not be unique. Since we are interested in the least solution, we address this problem by fixing each free variable to zero, so that we obtain a reduced system with a unique solution. This can be easily done by inspecting the graph $\mathcal{G}(\mathcal{M})$: all variables with zero probability of reaching G are detected by checking that they cannot be reached from any state in G in the reverse graph of $\mathcal{G}(\mathcal{M})$.

Regarding the λ-discrepancy for a coupling $\mathcal{C}$, if $\lambda = 1$, one can directly instantiate the aforementioned method with $G = \{(s,t) \in S \times S \mid \ell(s) \neq \ell(t)\}$ and $S_? = (S \times S) \setminus G$. As for generic $\lambda \in (0,1]$, the discrepancy $\gamma_\lambda^{\mathcal{C}}$ can be formulated as the least solution in $[0,1]^{S \times S}$ of the linear equation system

$$\boldsymbol{x} = \lambda \boldsymbol{A} \boldsymbol{x} + \lambda \boldsymbol{b}\,. \tag{3}$$

Remark 9. If one is interested in computing the λ-discrepancy for a particular pair of states (s,t), the method above can be applied on the least independent subsystem of Equation (3) containing the variable $x_{(s,t)}$. Moreover, assuming that for some pairs the λ-discrepancy is already known, the goal set can be extended with all those pairs with λ-discrepancy greater than zero. ■

4.2 Greedy Search Strategy for Computing an Optimal Coupling

In this section, we give a greedy strategy for moving toward an optimal coupling starting from a given one. Then we provide sufficient and necessary conditions for a coupling, ensuring that its associated λ-discrepancy coincides with δ_λ.

Hereafter, we fix a coupling $\mathcal{C} = (S \times S, A \times A, \omega, l)$ for $\mathcal{M} = (S, A, \pi, \ell)$. Let $s, t \in S$ and μ be a matching for $(\pi(s,\cdot), \pi(t,\cdot))$. We denote by $\mathcal{C}[(s,t)/\mu]$ the coupling for $\mathcal{M}$ with the same labeling function of $\mathcal{C}$ and transition function ω' defined by $\omega'((u,v),\cdot) = \omega((u,v),\cdot)$, for all $(u,v) \neq (s,t)$, and $\omega'((s,t),\cdot) = \mu$.

Lemma 10. *Let $\mathcal{C}$ be a coupling for $\mathcal{M}$, $s,t \in S$, $\omega' \in \pi(s,\cdot) \otimes \pi(t,\cdot)$, and $\mathcal{D} = \mathcal{C}[(s,t)/\omega']$. If $\Gamma_\lambda^{\mathcal{D}}(\gamma_\lambda^{\mathcal{C}})(s,t) < \gamma_\lambda^{\mathcal{C}}(s,t)$, then $\gamma_\lambda^{\mathcal{D}} \sqsubset \gamma_\lambda^{\mathcal{C}}$.*

Lemma 10 states that $\mathcal{C}$ can be improved w.r.t. $\trianglelefteq_\lambda$ by updating its transition function at (s,t), if $\ell(s) = \ell(t)$ and there exists $\omega' \in \pi(s,\cdot) \otimes \pi(t,\cdot)$ such that

$$\textstyle\sum_{u,v \in S} \gamma_\lambda^{\mathcal{C}}(u,v) \cdot \omega'(u,v) < \sum_{u,v \in S} \gamma_\lambda^{\mathcal{C}}(u,v) \cdot \omega((s,t),(u,v)).$$

Notice that, an optimal schedule ω' for $TP(\gamma_\lambda^{\mathcal{C}}, \pi(s,\cdot), \pi(t,\cdot))$ enjoys the above condition, so that, the update $\mathcal{C}[(s,t)/\omega']$ improves $\mathcal{C}$. This gives us a strategy for moving toward δ_λ by successive improvements on the couplings.

Now we proceed giving a sufficient and necessary condition for termination.

Lemma 11. *For any $\lambda \in (0,1]$, if $\gamma_\lambda^{\mathcal{C}} \neq \delta_\lambda$, then there exist $s,t \in S$ and a coupling $\mathcal{D} = \mathcal{C}[(s,t)/\omega']$ for $\mathcal{M}$ such that $\Gamma_\lambda^{\mathcal{D}}(\gamma_\lambda^{\mathcal{C}})(s,t) < \gamma_\lambda^{\mathcal{C}}(s,t)$.*

The above result ensures that, unless $\mathcal{C}$ is optimal w.r.t $\trianglelefteq_\lambda$, the hypothesis of Lemma 10 are satisfied, so that, we can further improve $\mathcal{C}$ as aforesaid.

The next statement proves that this search strategy is correct.

Theorem 12. *$\delta_\lambda = \gamma_\lambda^{\mathcal{C}}$ iff there is no coupling $\mathcal{D}$ for $\mathcal{M}$ such that $\Gamma_\lambda^{\mathcal{D}}(\gamma_\lambda^{\mathcal{C}}) \sqsubset \gamma_\lambda^{\mathcal{C}}$.*

Proof. We prove: $\delta_\lambda \neq \gamma_\lambda^{\mathcal{C}}$ iff there exists $\mathcal{D}$ such that $\Gamma_\lambda^{\mathcal{D}}(\gamma_\lambda^{\mathcal{C}}) \sqsubset \gamma_\lambda^{\mathcal{C}}$. ($\Rightarrow$) Assume $\delta_\lambda \neq \gamma_\lambda^{\mathcal{C}}$. By Lemma 11, there are $s, t \in S$ and $\omega' \in \pi(s, \cdot) \otimes \pi(t, \cdot)$ such that $\lambda \cdot \sum_{u,v \in S} \gamma_\lambda^{\mathcal{C}}(u, v) \cdot \omega'(u, v) < \gamma_\lambda^{\mathcal{C}}(s, t)$. As in the proof of Lemma 10, we have that $\mathcal{D} = \mathcal{C}[(s, t)/\omega']$ satisflies $\Gamma^{\mathcal{D}}(\gamma_\lambda^{\mathcal{C}}) \sqsubset \gamma_\lambda^{\mathcal{C}}$. ($\Leftarrow$) Let $\mathcal{D}$ be such that $\Gamma_\lambda^{\mathcal{D}}(\gamma_\lambda^{\mathcal{C}}) \sqsubset \gamma_\lambda^{\mathcal{C}}$. By Tarski's fixed point theorem $\gamma_\lambda^{\mathcal{D}} \sqsubset \gamma_\lambda^{\mathcal{C}}$. By Theorem 8, $\delta_\lambda \sqsubseteq \gamma_\lambda^{\mathcal{D}} \sqsubset \gamma_\lambda^{\mathcal{C}}$. □

Remark 13. Note that, in general there could be an infinite number of couplings for a given MC. However, for each fixed $d \in [0, 1]^{S \times S}$, the linear function mapping $\omega((s, t), \cdot)$ to $\lambda \sum_{u,v \in S} d(u, v) \cdot \omega((s, t), (u, v))$ achieves its minimum at some vertex in the transportation polytope $\pi(s, \cdot) \otimes \pi(t, \cdot)$. Since the number of such vertices are finite, using the optimal TP schedule for the update, ensures that the search strategy is always terminating. ■

5 The On-the-Fly Algorithm

In this section we provide an on-the-fly algorithm for exact computation of the bisimilarity distance δ_λ for generic discount factors making full use of the greedy strategy presented in Section 4.2.

Let $Q \subseteq S \times S$. Assume we want to compute $\delta_\lambda(s, t)$, for all $(s, t) \in Q$. The method proposed in Section 4.2 has the following key features:

1. the improvement of each coupling $\mathcal{C}$ is obtained by a *local* update of its transition function at some state (u, v) in $\mathcal{C}$;
2. the strategy does not depend on the choice of the state (u, v);
3. whenever a coupling $\mathcal{C}$ is considered, the over-approximation $\gamma_\lambda^{\mathcal{C}}$ of the distance can be computed by solving a system of linear equations.

Among them, only the last one requires a visit of the coupling. However, as noticed in Remark 9, the value $\gamma_\lambda^{\mathcal{C}}(s, t)$ can be computed without considering the entire linear system of Equation (3), but only its smallest independent subsystem containing the variable $x_{(s,t)}$, which is obtained by restricting on the variables $x_{(u,v)}$ such that $(u, v) \in R_C((s, t))$. This subsystem can be further reduced, by Gaussian elimination, when some values for δ_λ are known. The last observation suggests that, in order to compute $\gamma_\lambda^{\mathcal{C}}(s, t)$, we do not need to store the entire coupling, but it can be constructed on-the-fly.

The exact computation of the bisimilarity pseudometric is implemented by Algorithm 1. It takes as input a finite MC $\mathcal{M} = (S, A, \pi, \ell)$, a discount factor λ, and a query set Q. We assume the following global variables to store: $\mathcal{C}$, the current partial coupling; d, the λ-discrepancy associated with $\mathcal{C}$; $ToCompute$, the pairs of states for which the distance has to be computed; $Exact$, the pairs of states (s, t) such that $d(s, t) = \delta_\lambda(s, t)$; $Visited$, the states of $\mathcal{C}$ considered so far. At the beginning (line 1) both the coupling $\mathcal{C}$ and the discrepancy d are empty, there are no visited states, and no exact computed distances. While there are still pairs left to be computed (line 2), we pick one (line 3), say (s, t). According to the definition of δ_λ, if $\ell(s) \neq \ell(t)$ then $\delta_\lambda(s, t) = 1$; if $s = t$ then $\delta_\lambda(s, t) = 0$, so

Algorithm 1. On-the-Fly Bisimilarity Pseudometric

```
Input: MC $\mathcal{M} = (S, A, \pi, \ell)$; discount factor $\lambda \in (0, 1]$; query $Q \subseteq S \times S$.
 1. $\mathcal{C} \leftarrow$ empty; $d \leftarrow$ empty; $Visited \leftarrow \emptyset$; $Exact \leftarrow \emptyset$; $ToCompute \leftarrow Q$;   // Init.
 2. while $ToCompute \neq \emptyset$ do
 3.   pick $(s, t) \in ToCompute$
 4.   if $\ell(s) \neq \ell(t)$ then
 5.     $d(s, t) \leftarrow 1$; $Exact \leftarrow Exact \cup \{(s, t)\}$; $Visited \leftarrow Visited \cup \{(s, t)\}$
 6.   else if $s = t$ then
 7.     $d(s, t) \leftarrow 0$; $Exact \leftarrow Exact \cup \{(s, t)\}$; $Visited \leftarrow Visited \cup \{(s, t)\}$
 8.   else   // if $(s, t)$ is nontrivial
 9.     if $(s, t) \notin Visited$ then pick $\omega \in \pi(s, \cdot) \otimes \pi(t, \cdot)$; $SetPair(\mathcal{M}, (s, t), \omega)$
10.     $Discrepancy(\lambda, (s, t))$   // update $d$ as the $\lambda$-discrepancy for $\mathcal{C}$
11.     while $\exists (u, v) \in R_{\mathcal{C}}((s, t))$. $\mathcal{C}[(u, v)]$ not opt. for $TP(d, \pi(u, \cdot), \pi(v, \cdot))$ do
12.       $\omega \leftarrow$ optimal schedule for $TP(d, \pi(u, \cdot), \pi(v, \cdot))$
13.       $SetPair(\mathcal{M}, (u, v), \omega)$   // improve the current coupling
14.       $Discrepancy(\lambda, (s, t))$   // update $d$ as the $\lambda$-discrepancy for $\mathcal{C}$
15.     end while
16.     $Exact \leftarrow Exact \cup R_{\mathcal{C}}((s, t))$   // add new exact distances
17.     remove from $\mathcal{C}$ all edges exiting from nodes in $Exact$
18.   end if
19.   $ToCompute \leftarrow ToCompute \setminus Exact$   // remove exactly computed pairs
20. end while
21. return $d{\restriction}_Q$   // return the distance for all pairs in $Q$
```

that, $d(s, t)$ is set accordingly, and (s, t) is added to *Exact* (lines 4–7). Otherwise, if (s, t) was not previously visited, a matching $\omega \in \pi(s, \cdot) \otimes \pi(t, \cdot)$ is guessed, and the routine *SetPair* updates the coupling $\mathcal{C}$ at (s, t) with ω (line 9), then the routine *Discrepancy* updates d with the λ-discrepancy associated with $\mathcal{C}$ (line 10). According to the greedy strategy, $\mathcal{C}$ is successively improved and d is consequently updated, until no further improvements are possible (lines 11–15). Each improvement is demanded by the existence of a better schedule for $TP(d, \pi(u, \cdot), \pi(u, \cdot))$ (line 11). Note that, each improvement actually affects the current value of $d(s, t)$. This is done by restricting our attention only to the pairs that are reachable from (s, t) in $\mathcal{G}(\mathcal{C})$. It is worth to note that $\mathcal{C}$ is constantly updated, hence $R_{\mathcal{C}}((s, t))$ may differ from one iteration to another. When line 16 is reached, for each $(u, v) \in R_{\mathcal{C}}((s, t))$, we are guaranteed that $d(u, v) = \delta_{\lambda}(s, t)$, therefore $R_{\mathcal{C}}((s, t))$ is added to *Exact*, and these values can be used in successive computations, so the edges exiting from these states are removed from $\mathcal{G}(\mathcal{C})$. In line 19, the exact pairs computed so far are removed from *ToCompute*. Finally, if no more pairs need be considered, the exact distance on Q is returned (line 21).

Algorithm 1 calls the subroutines *SetPair* and *Discrepancy*, respectively, to construct/update the coupling $\mathcal{C}$, and to update the current over-approximation d during the computation. Now we explain how they work.

SetPair (Algorithm 2) takes as input an MC $\mathcal{M} = (S, A, \pi, \ell)$, a pair of states (s, t), and a matching $\omega \in \pi(s, \cdot) \otimes \pi(t, \cdot)$. In lines 1–2 the transition function of the coupling $\mathcal{C}$ is set to ω at (s, t), then (s, t) is added to *Visited*. The on-the-fly

Algorithm 2. *SetPair*$(\mathcal{M}, (s,t), \omega)$

```
Input: MC $\mathcal{M} = (S, A, \pi, \ell)$; $s, t \in S$; $\omega \in \pi(s,\cdot) \otimes \pi(t,\cdot)$
 1. $\mathcal{C}[(s,t)] \leftarrow \omega$    // update the coupling at $(s,t)$ with $\omega$
 2. $Visited \leftarrow Visited \cup \{(s,t)\}$    // set $(s,t)$ as visited
 3. for all $(u,v) \in \{(u',v') \mid \omega(u',v') > 0\} \setminus Visited$ do    // for all demanded pairs
 4.    $Visited \leftarrow Visited \cup \{(u,v)\}$
 5.    if $u = v$ then $d(u,v) \leftarrow 0$; $Exact \leftarrow Exact \cup \{(u,v)\}$;
 6.    if $\ell(u) \neq \ell(v)$ then $d(u,v) \leftarrow 1$; $Exact \leftarrow Exact \cup \{(u,v)\}$;
 7.    // propagate the construction
 8.    if $(u,v) \notin Exact$ then
 9.       pick $\omega' \in \pi(u,\cdot) \otimes \pi(v,\cdot)$    // guess a matching
10.       $SetPair(\mathcal{M}, (u,v), \omega')$
11.    end if
12. end for
```

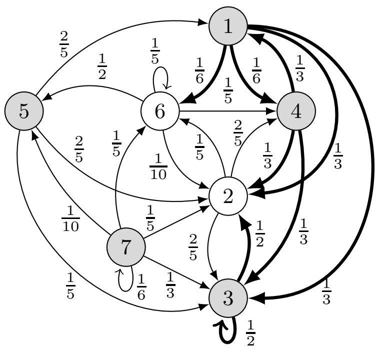
Algorithm 3. *Discrepancy*$(\lambda, (s,t))$

```
Input: discount factor $\lambda \in (0,1]$; $s, t \in S$
 1. $Nonzero \leftarrow \emptyset$    // detect non-zero variables
 2. for all $(u,v) \in R_{\mathcal{C}}((s,t)) \cap Exact$ such that $d(u,v) > 0$ do
 3.    $Nonzero \leftarrow Nonzero \cup \{(u',v') \mid (u,v) \rightsquigarrow (u',v')$ in $\mathcal{G}^{-1}(\mathcal{C})\}$
 4. end for
 5. for all $(u,v) \in R_{\mathcal{C}}((s,t)) \setminus Nonzero$ do    // set distance to zero
 6.    $d(u,v) \leftarrow 0$; $Exact \leftarrow Exact \cup \{(u,v)\}$
 7. end for
 8. // construct the reduced linear system over nonzero variables
 9. $\boldsymbol{A} \leftarrow (\mathcal{C}[(u,v)](u',v'))_{(u,v),(u',v') \in Nonzero}$
10. $\boldsymbol{b} \leftarrow \left(\sum_{(u',v') \in Exact} d(u',v') \cdot \mathcal{C}[(u,v)](u',v')\right)_{(u,v) \in Nonzero}$
11. $\tilde{\boldsymbol{x}} \leftarrow$ solve $\boldsymbol{x} = \lambda \boldsymbol{A}\boldsymbol{x} + \lambda \boldsymbol{b}'$    // solve the reduced linear system
12. for all $(u,v) \in Nonzero$ do    // update distances
13.    $d(u,v) \leftarrow \tilde{x}_{(u,v)}$
14. end for
```

construction of the coupling is recursively propagated to the successors of (s,t) in $\mathcal{G}(\mathcal{C})$. During this construction, if some states with trivial distances are encountered, d and *Exact* are updated accordingly (lines 5–6).

Discrepancy (Algorithm 3) takes as input a discount factor λ and a pair of states (s,t). It constructs the smallest (reduced) independent subsystem of Equation 3 having the variable $x_{(s,t)}$ (lines 9–10). As noticed in Remark 9, the least solution is computed by fixing d to zero for all the pairs which cannot be reached from any pair in *Exact* and such that its distance is greater than zero (lines 5–7). Then, the discrepancy is computed and d is consequently updated.

Next, we present a simple example of Algorithm 1, showing the main features of our method: (1) the on-the-fly construction of the (partial) coupling, and (2) the restriction only to those variables which are demanded for the solution of the system of linear equations.



$\mathcal{C}_0$

(1,4)	**1**	**2**	**3**	
2	1/3			1/3
3		1/3		1/3
4			1/6	1/6
6			1/6	1/6
	1/3	1/3	1/3	

(3,4)	**1**	**2**	**3**	
2	1/3	1/6		1/2
3		1/6	1/3	1/2
	1/3	1/3	1/3	

$\mathcal{C}_1$

(1,4)	**1**	**2**	**3**	
2		1/3		1/3
3			1/3	1/3
4	1/6			1/6
6	1/6			1/6
	1/3	1/3	1/3	

Fig. 1. Execution trace for the computation of $\delta_1(1,4)$ (details in Example 14)

Example 14 (On-the-fly computation). Consider the undiscounted distance between states 1 and 4 for the $\{\text{white}, \text{gray}\}$-labeled MC depicted in Figure 1.

Algorithm 1 guesses an initial coupling $\mathcal{C}_0$ with transition distribution ω_0. This is done considering only the pairs of states which are needed: starting from $(1,4)$, the distribution $\omega_0((1,4),\cdot)$ is guessed as in Figure 1, which demands for the exploration of $(3,4)$ and a guess $\omega_0((3,4),\cdot)$. Since no other pairs are demanded, the construction of $\mathcal{C}_0$ terminates. This gives the equation system:

$$\begin{cases} x_{1,4} = \frac{1}{3}\cdot\overbrace{x_{1,2}}^{=1} + \frac{1}{3}\cdot\overbrace{x_{2,3}}^{=1} + \frac{1}{6}\cdot x_{3,4} + \frac{1}{6}\cdot\overbrace{x_{3,6}}^{=1} = \frac{1}{6}\cdot x_{3,4} + \frac{5}{6} \\ x_{3,4} = \frac{1}{3}\cdot\overbrace{x_{1,2}}^{=1} + \frac{1}{6}\cdot\overbrace{x_{2,2}}^{=0} + \frac{1}{6}\cdot\overbrace{x_{2,3}}^{=1} + \frac{1}{3}\cdot\overbrace{x_{3,3}}^{=0} = \frac{1}{2}\,. \end{cases}$$

Note that the only variables appearing in the above equation system correspond to the pairs which have been considered so far. The least solution for it is given by $d^{\mathcal{C}_0}(1,4) = \frac{11}{12}$ and $d^{\mathcal{C}_0}(3,4) = \frac{1}{2}$.

Now, these solutions are taken as the costs of a TP, from which we get an optimal transportation schedule $\omega_1((1,4),\cdot)$ improving $\omega_0((1,4),\cdot)$. The distribution ω_1 is used to update $\mathcal{C}_0$ to $\mathcal{C}_1 = \mathcal{C}_0[(1,4)/\omega_1]$ (depicted in Figure 1), obtaining the following new equation system:

$$x_{1,4} = \frac{1}{3}\cdot\overbrace{x_{2,2}}^{=0} + \frac{1}{3}\cdot\overbrace{x_{3,3}}^{=0} + \frac{1}{6}\cdot x_{1,4} + \frac{1}{6}\cdot\overbrace{x_{1,6}}^{=1} = \frac{1}{6}\cdot x_{1,4} + \frac{1}{6}\,,$$

which has $d^{\mathcal{C}_1}(1,4) = \frac{1}{5}$ as least solution. Note that, $(3,4)$ is no more demanded, thus we do not need to update it. Running again the TP on the improved over-approximation $d^{\mathcal{C}_1}$, we discover that the coupling $\mathcal{C}_1$ cannot be further improved, hence we stop the computation, returning $\delta_1(1,4) = d^{\mathcal{C}_1}(1,4) = \frac{1}{5}$.

It is worth noticing that Algorithm 1 does not explore the entire MC, not even all the reachable states from 1 and 4. The only edges in the MC which have been considered during the computation are highlighted in Figure 1. ■

Table 1. Comparison between the on-the-fly algorithm and the iterative method

# States	On-the-Fly (exact)		Iterating (approximated)			Approximation
	Time (s)	# TPs	Time (s)	# Iterations	# TPs	Error
5	0.019675	1.19167	0.0389417	1.73333	26.7333	0.139107
6	0.05954	3.04667	0.09272	1.82667	38.1333	0.145729
7	0.13805	6.01111	0.204789	2.19444	61.7278	0.122683
8	0.255067	8.5619	0.364019	2.30476	83.0286	0.11708
9	0.499983	12.0417	0.673275	2.57917	114.729	0.111104
10	1.00313	18.7333	1.27294	3.11111	174.363	0.0946047
11	2.15989	25.9733	2.66169	3.55667	239.557	0.0959714
12	4.64225	34.797	5.52232	4.04242	318.606	0.0865612
13	6.73513	39.9582	8.06186	4.63344	421.675	0.0977743
14	6.33637	38.0048	7.18807	4.91429	593.981	0.118971
17	11.2615	47.0143	12.8048	5.88571	908.61	0.13213
19	26.6355	61.1714	29.6542	6.9619	1328.6	0.14013
20	34.379	66.4571	38.2058	7.5381	1597.92	0.142834

Remark 15. Notably, Algorithm 1 can also be used for computing over-approximated distances. Indeed, assuming over-estimates for some particular distances are already known, they can be taken as inputs and used in our algorithm simply storing them in the variable d and treated as "exact" values. In this way our method will return the least over-approximation of the distance agreeing with the given over-estimates. This modification of the algorithm can be used to further decrease the exploration of the MC. Moreover, it can be employed in combination with other existing approximated algorithms, having the advantage of an on-the-fly state space exploration. ∎

6 Experimental Results

In this section, we evaluate the performances of the on-the-fly algorithm on a collection of randomly generated MCs[1].

First, we compare the execution times of the on-the-fly algorithm with those of the iterative method proposed in [4] in the discounted case. Since the iterative method only allows for the computation of the distance for all state pairs at once, the comparison is (in fairness) made with respect to runs of our on-the-fly algorithm with input query being the set of all state pairs. For each input instance, the comparison involves the following steps:

1. we run the on-the-fly algorithm, storing both execution time and the number of solved transportation problems,

[1] The tests have been made using a prototype implementation coded in Mathematica® (available at `http://people.cs.aau.dk/~mardare/projects/tools/mc_dist.zip`) running on an Intel Core-i7 3.4 GHz processor with 12GB of RAM.

Table 2. Average performances of the on-the-fly algorithm on single-pair queries. In the first to columns the out-degree is 3; in the last two columns, the out-degree varies from 2 to # States. (*) For 20, 30 and 50 states, out-degree is 4.

# States	out-degree = 3		$2 \leq$ out-degree $\leq$ # States*	
	Time (s)	# TPs	Time (s)	# TPs
5	0.00594318	0.272727	0.011654	0.657012
6	0.0115532	0.548936	0.0304482	1.66696
7	0.0168408	0.980892	0.0884878	3.67706
8	0.0247971	1.34606	0.164227	5.30112
9	0.0259426	1.29074	0.394543	8.16919
10	0.0583405	2.03887	1.1124	13.0961
11	0.0766988	1.82706	2.22016	18.7228
12	0.0428891	1.62038	4.94045	26.0965
13	0.06082	1.88134	10.3606	35.1738
14	0.0894778	2.79441	20.1233	46.0775
20	0.35631	6.36833	1.5266	13.1367
30	4.66113	17.3167	74.8146	76.2642
50	27.2147	30.8217	2234.54	225.394

2. then, on the same instance, we execute the iterative method until the running time exceeds that of step 1. We report the execution time, the number of iterations, and the number of solved transportation problems.
3. Finally, we calculate the approximation error between the exact solution δ_λ computed by our method at step 1 and the approximate result d obtained in step 2 by the iterative method, as $\max_{s,t \in S} \delta_\lambda(s,t) - d(s,t)$.

This has been made on a collection of MCs varying from 5 to 20 states. For each $n = 5, \ldots, 20$, we have considered 80 randomly generated MCs per out-degree, varying from 2 to n. Table 1 reports the average results of the comparison.

As can be seen, our use of a greedy strategy in the construction of the couplings leads to a significant improvement in the performances. We are able to compute the exact solution before the iterative method can under-approximate it with an error of ≈ 0.1, which is a considerable error for a value in $[0, 1]$.

So far, we only examined the case when the on-the-fly algorithm is run on all state pairs at once. Now, we show how the performance of our method is improved even further when the distance is computed only for single pairs of states. Table 2 shows the average execution times and number of solved transportation problems for (nontrivial) single-pair queries for randomly generated of MCs with number of states varying from 5 to 50. In the first two columns we consider MCs with out-degree equal to 3, while the last two columns show the average values for out-degrees varying from 2 to the number of states of the MCs. The results show that, when the out-degree of the MCs is low, our algorithm performs orders of magnitude better than in the general case. This is illustrated in Figure 2, where the distributions of the execution times for out-degree 6 and 8 are juxtaposed, in the case of MCs with 14 states. Each bar in the histogram

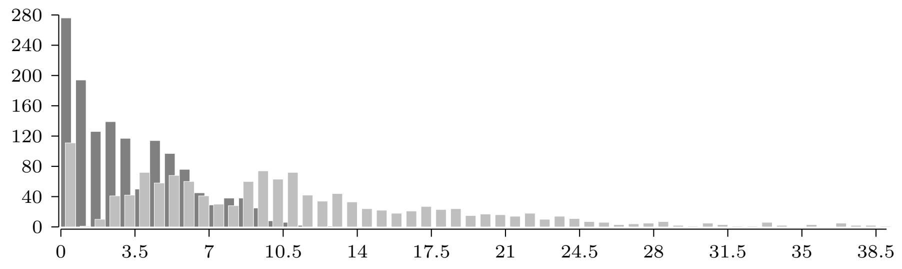


Fig. 2. Distribution of the execution times (in seconds) for 1332 tests on randomly generated MCs with 14 states, out-degree 6 (darkest) and 8 (lightest)

represents the number of tests that terminate within the time interval indicated in the x-axis.

Notably, our method may perform better on large queries than on single-pairs queries. This is due to the fact that, although the returned value does not depend on the order the queried pairs are considered, a different order may speed up the performances. When the algorithm is run on more than a single pair, the way they are picked may increase the performances (e.g., compare the execution times in Tables 1 and 2 for MCs with 14 states).

7 Conclusions and Future Work

In this paper we have proposed an on-the-fly algorithm for computing exactly the bisimilarity distance between Markov chains, introduced by Desharnais et al. in [8]. Our algorithm represents an important improvement of the state of the art in this field where, before our contribution, the known tools were only concerned with computing approximations of the bisimilarity distances and they were, in general, based on iterative techniques. We demonstrate that, using on-the-fly techniques, we cannot only calculate exactly the bisimilarity distance, but the computation time is improved with orders of magnitude with respect to the corresponding iterative approaches. Moreover, our technique allows for the computation on a set of target distances that might be done by only investigating a significantly reduced set of states, and for further improvement of speed.

Our algorithm can be practically used to address a large spectrum of problems. For instance, it can be seen as a method to decide whether two states of a given MC are probabilistic bisimilar, to identify bisimilarity classes, or to solve lumpability problems. It is sufficiently robust to be used with approximation techniques as, for instance, to provide a least over-approximation of the behavioural distance given over-estimates of some particular distances. It can be integrated with other approximate algorithms, having the advantage of the efficient on-the-fly state space exploration.

Having a practically efficient tool to compute bisimilarity distances opens the perspective of new applications already announced in previous research papers. One of these is the state space reduction problem for MCs. Our technique can be used in this context as an indicator for the sets of neighbour states that can be collapsed due to their similarity; it also provides a tool to estimate the difference between the initial MC and the reduced one, hence a tool for the approximation theory of Markov chains.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
2. Cai, X., Gu, Y.: Measuring Anonymity. In: Bao, F., Li, H., Wang, G. (eds.) ISPEC 2009. LNCS, vol. 5451, pp. 183–194. Springer, Heidelberg (2009)
3. Chatterjee, K., de Alfaro, L., Majumdar, R., Raman, V.: Algorithms for Game Metrics. Logical Methods in Computer Science 6(3) (2010)
4. Chen, D., van Breugel, F., Worrell, J.: On the Complexity of Computing Probabilistic Bisimilarity. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 437–451. Springer, Heidelberg (2012)
5. Comanici, G., Panangaden, P., Precup, D.: On-the-Fly Algorithms for Bisimulation Metrics. In: Proceedings of the 9th International Conference on Quantitative Analysis of Systems, QEST, September 17-20, pp. 94–103 (2012)
6. Comanici, G., Precup, D.: Basis function discovery using spectral clustering and bisimulation metrics. In: AAMAS 2011, vol. 3, pp. 1079–1080. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2011)
7. Dantzig, G.B.: Application of the Simplex method to a transportation problem. In: Koopmans, T. (ed.) Activity Analysis of Production and Allocation, pp. 359–373. J. Wiley, New York (1951)
8. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled Markov processes. Theoretical Computer Science 318(3), 323–354 (2004)
9. Ferns, N., Panangaden, P., Precup, D.: Metrics for finite Markov Decision Processes. In: Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence, UAI, pp. 162–169. AUAI Press (2004)
10. Ford, L.R., Fulkerson, D.R.: Solving the Transportation Problem. Management Science 3(1), 24–32 (1956)
11. Griffeath, D.: A maximal coupling for markov chains. Probability Theory and Related Fields 31, 95–106 (1975)
12. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
13. Mitzenmacher, M., Upfal, E.: Probability and Computing - randomized algorithms and probabilistic analysis. Cambridge University Press (2005)
14. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons, Inc., New York (1986)
15. Thorsley, D., Klavins, E.: Approximating stochastic biochemical processes with Wasserstein pseudometrics. IET Systems Biology 4(3), 193–211 (2010)
16. van Breugel, F., Sharma, B., Worrell, J.: Approximating a Behavioural Pseudometric without Discount for Probabilistic Systems. Logical Methods in Computer Science 4(2), 1–23 (2008)
17. van Breugel, F., Worrell, J.: Approximating and computing behavioural distances in probabilistic transition systems. Theoretical Computer Science 360(1-3), 373–385 (2006)

The Quest for Minimal Quotients for Probabilistic Automata

Christian Eisentraut[1], Holger Hermanns[1], Johann Schuster[2], Andrea Turrini[1], and Lijun Zhang[3,1]

[1] Department of Computer Science, Saarland University, Germany
[2] Department of Computer Science, University of the Federal Armed Forces Munich, Germany
[3] DTU Informatics, Technical University of Denmark, Denmark

Abstract. One of the prevailing ideas in applied concurrency theory and verification is the concept of automata minimization with respect to strong or weak bisimilarity. The minimal automata can be seen as canonical representations of the behaviour modulo the bisimilarity considered. Together with congruence results wrt. process algebraic operators, this can be exploited to alleviate the notorious state space explosion problem. In this paper, we aim at identifying minimal automata and canonical representations for concurrent probabilistic models. We present minimality and canonicity results for probabilistic automata wrt. strong and weak bisimilarity, together with polynomial time minimization algorithms.

1 Introduction

Markov decision processes (*MDP*s) are models appearing in areas such as operations research, automated planning, and decision support systems. In the concurrent systems context, they arise in the form of probabilistic automata (*PA*s) [17]. *PA*s form the backbone model of successful model checkers such as PRISM [12] enabling the analysis of randomised concurrent systems. Despite the remarkable versatility of this approach, its power is limited by the state space explosion problem, and several approaches are being pursued to alleviate that problem.

In related fields, a favourable strategy is to minimize the system – or components thereof – to the quotient under bisimilarity. This can speed up the overall model analysis or turn a too large problem into a tractable one [2,4,9]. Both, strong and weak bisimilarity are used in practice, with weaker relations leading to greater reduction. However, this approach has never been explored in the context of *MDP*s or probabilistic automata. One reason is that thus far no effective decision algorithm was at hand for weak bisimilarity on *PA*s. A polynomial time algorithm has been proposed only recently [10]. But that algorithm is a decision algorithm, not a minimization algorithm. This paper therefore focusses on a seemingly tiny problem: Does there exist a *unique minimal* representative of a given probabilistic automaton with respect to weak bisimilarity? And can we compute it? In fact, this turns out to be an intricate problem. We nevertheless arrive at a polynomial time algorithm.

Notably, minimality with respect to the number of states of a probabilistic automaton does not imply minimality with respect to the number of transitions. And further minimization is possible with respect to transition fanouts, the latter referring to the number of target states of a transition with non-zero probability. The minimality of an automaton thus needs to be considered with respect to all the three characteristics: number of states, of transitions and of transitions' fanouts.

We consider our results as a breakthrough with wide ranging consequences. Since weak probabilistic bisimilarity enjoys congruence properties for parallel composition and hiding on *PA*s, compositional minimization approaches can now be carried out efficiently. And because *PA*s comprise *MDP*s, we think it is not far fetched to imagine fruitful applications in areas such as operations research, automated planning, and decision support systems.

As a byproduct, our results provide tailored algorithms for strong probabilistic bisimilarity on *PA*s and strong and weak bisimilarity on labelled transition systems.

Organization. After the preliminaries in Section 2, we recall the bisimulation relations in Section 3 and we introduce the preorders between automata in Section 4. Then we present automaton reductions in Section 5 which will be used to compute the normal forms defined in Section 6. We conclude the paper in Section 7 with some remarks.

2 Preliminaries

Sets, Relations and Distributions: Given sets X, Y, and Z and relations $\mathcal{R} \subseteq X \times Y$ and $\mathcal{S} \subseteq Y \times Z$, we denote by $\mathcal{R} \circ \mathcal{S}$ the relation $\mathcal{R} \circ \mathcal{S} \subseteq X \times Z$ such that $\mathcal{R} \circ \mathcal{S} = \{ (x, z) \mid \exists y \in Y. x \mathrel{\mathcal{R}} y \wedge y \mathrel{\mathcal{S}} z \}$.

For a set X, we denote by $\mathrm{SubDisc}(X)$ the set of discrete sub-probability distributions over X. Given $\rho \in \mathrm{SubDisc}(X)$, we denote by $|\rho|$ the size $\rho(X) = \sum_{s \in X} \rho(s)$ of a distribution. We call a distribution ρ *full*, or simply a *probability* distribution, if $|\rho| = 1$. The set of all discrete probability distributions over X is denoted by $\mathrm{Disc}(X)$. Given $\rho \in \mathrm{SubDisc}(X)$, we denote by $\mathrm{Supp}(\rho)$ the set $\{ x \in X \mid \rho(x) > 0 \}$, by $\rho(\perp)$ the value $1 - \rho(X)$ where $\perp \notin X$, by δ_x the *Dirac* distribution such that $\rho(x) = 1$ for $x \in X \cup \{\perp\}$ where $\delta_\perp$ represents the empty distribution such that $\rho(X) = 0$. For a constant $c \geq 0$, we denote by $c\rho$ the distribution defined by $(c\rho)(x) = c \cdot \rho(x)$ if $c|\rho| \leq 1$. Further, for $\rho \in \mathrm{Disc}(X)$ and $x \in X$ such that $\rho(x) < 1$, we denote by $\rho \backslash x$ the *rescaled* distribution such that $(\rho \backslash x)(y) = \frac{\rho(y)}{1 - \rho(x)}$ if $y \neq x$, 0 otherwise. We define the distribution $\rho = \rho_1 \oplus \rho_2$ by $\rho(s) = \rho_1(s) + \rho_2(s)$ provided $|\rho| \leq 1$, and conversely we say ρ can be split into ρ_1 and ρ_2. Since $\oplus$ is associative and commutative, we may use the notation $\bigoplus$ for arbitrary finite sums.

The lifting $\mathcal{L}(\mathcal{R}) \subseteq \mathrm{Disc}(X) \times \mathrm{Disc}(X)$ [13] of an equivalence relation $\mathcal{R}$ on X is defined as: for $\rho_1, \rho_2 \in \mathrm{Disc}(X)$, $\rho_1 \mathrel{\mathcal{L}(\mathcal{R})} \rho_2$ if and only if for each $\mathcal{C} \in X/_{\mathcal{R}}$, $\rho_1(\mathcal{C}) = \rho_2(\mathcal{C})$, where $X/_{\mathcal{R}} = \{ [x]_{\mathcal{R}} \mid x \in X \}$ and $[x]_{\mathcal{R}} = \{ x' \in X \mid x' \mathrel{\mathcal{R}} x \}$.

Models: A probabilistic automaton (*PA*) $\mathcal{A}$ is a tuple $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$, where S is a countable set of *states*, $\bar{s} \in S$ is the *start* state, Σ is a countable set of *actions*, and $\mathcal{T} \subseteq S \times \Sigma \times \mathrm{Disc}(S)$ is a *transition relation*. In this work we consider only finite PAs, i.e., automata such that S and $\mathcal{T}$ are finite.

An example of *PA* is sketched in Figure 1(a), the precise probabilities are left unspecified, and Dirac transitions directly connect states. The set Σ is partitioned into two sets $H = \{\tau\}$ and E of internal (hidden) and external actions, respectively; we refer to

$\bar{s}$ also as the *initial* state and we let s,t,u,v, and their variants with indices range over S and a, b range over Σ.

A transition $tr = (s, a, \nu) \in \mathcal{T}$, also denoted by $s \xrightarrow{a} \nu$, is said to *leave* from state s, to be *labelled* by a, and to *lead* to ν, also denoted by ν_{tr}. We denote by $src(tr)$ the *source* state s, by $act(tr)$ the *action* a, and by $trg(tr)$ the *target* distribution ν. We also say that s enables action a, that action a is enabled from s, and that (s, a, ν) is enabled from s. Finally, we denote by $\mathcal{T}(s)$ the set of transitions enabled from s, i.e., $\mathcal{T}(s) = \{\, tr \in \mathcal{T} \mid src(tr) = s \,\}$, and similarly for $a \in \Sigma$, by $\mathcal{T}(a)$ the set of transitions with action a, i.e., $\mathcal{T}(a) = \{\, tr \in \mathcal{T} \mid act(tr) = a \,\}$.

Given a state s, an action a and a countable set of indices I, we say that there exists a *combined transition* $s \xrightarrow{a}_{\mathrm{c}} \nu$ if there exist a family of transitions $\{(s, a, \nu_i) \in \mathcal{T}\}_{i \in I}$ and a family $\{c_i \in \mathbb{R}_{\geq 0}\}_{i \in I}$ such that $\sum_{i \in I} c_i = 1$ and $\nu = \bigoplus_{i \in I} c_i \nu_i$.

We call a *PA* $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$ a Labelled Transition System (*LTS*), if $(s, a, \mu) \in \mathcal{T}$ implies $\mu = \delta_t$ for some $t \in S$.

Weak Transitions: An *execution fragment* α of a *PA* $\mathcal{A}$ is a finite or infinite sequence of alternating states and actions $\alpha = s_0 a_1 s_1 a_2 s_2 \dots$ starting from a state $\mathit{first}(\alpha) = s_0$ and, if the sequence is finite, ending with a state $\mathit{last}(\alpha)$, such that for each $i > 0$ there exists $(s_{i-1}, a_i, \nu_i) \in \mathcal{T}$ such that $\nu_i(s_i) > 0$. The *length* of α, denoted by $|\alpha|$, is the number of occurrences of actions in α. If α is infinite, then $|\alpha| = \infty$. Denote by $\mathit{frags}(\mathcal{A})$ the set of execution fragments of $\mathcal{A}$ and by $\mathit{frags}^*(\mathcal{A})$ the set of finite execution fragments of $\mathcal{A}$. An execution fragment α is a *prefix* of an execution fragment α', denoted by $\alpha \leqslant \alpha'$, if the sequence α is a prefix of the sequence α'. The *trace* of α, denoted by $\mathit{trace}(\alpha)$, is the sub-sequence of external actions of α; we denote by ε the empty trace. Similarly, we define $\mathit{trace}(a) = a$ for $a \in E$ and $\mathit{trace}(\tau) = \varepsilon$.

Given a *PA* $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$, the *reachable fragment* of $\mathcal{A}$ is the *PA* $RF(\mathcal{A}) = (S', \bar{s}, \Sigma, \mathcal{T}')$ where $S' = \{\, s \in S \mid \exists \alpha \in \mathit{frags}^*(\mathcal{A}).\mathit{first}(\alpha) = \bar{s} \wedge \mathit{last}(\alpha) = s \,\}$ and $\mathcal{T}' = \{\, (s, a, \nu) \in \mathcal{T} \mid s \in S' \,\}$.

A *scheduler* for a *PA* $\mathcal{A}$ is a function $\sigma\colon \mathit{frags}^*(\mathcal{A}) \to \mathrm{SubDisc}(\mathcal{T})$ such that for each finite execution fragment α, $\sigma(\alpha) \in \mathrm{SubDisc}(\mathcal{T}(\mathit{last}(\alpha)))$. A scheduler is *Dirac* if it assigns a Dirac distribution to each execution fragment and it is *determinate* if for each pair of execution fragments α, α', $\mathit{trace}(\alpha) = \mathit{trace}(\alpha')$ and $\mathit{last}(\alpha) = \mathit{last}(\alpha')$ imply that $\sigma(\alpha) = \sigma(\alpha')$. It is worthwhile to note that a determinate scheduler satisfies $\sigma(\alpha) = \sigma(\mathit{last}(\alpha))$ when $\mathit{trace}(\alpha) = \varepsilon$.

Given a scheduler σ and a finite execution fragment α, the distribution $\sigma(\alpha)$ describes how transitions are chosen to move on from $\mathit{last}(\alpha)$. A scheduler σ and a state s induce a probability distribution $\nu_{\sigma,s}$ over execution fragments as follows. The basic measurable events are the cones of finite execution fragments, where the cone of a finite execution fragment α, denoted by C_α, is the set $\{\, \alpha' \in \mathit{frags}(\mathcal{A}) \mid \alpha \leqslant \alpha' \,\}$. The probability $\nu_{\sigma,s}$ of a cone C_α is defined recursively as follows:

$$\nu_{\sigma,s}(C_\alpha) = \begin{cases} 0 & \text{if } \alpha = t \text{ for a state } t \neq s, \\ 1 & \text{if } \alpha = s, \\ \nu_{\sigma,s}(C_{\alpha'}) \cdot \sum_{tr \in \mathcal{T}(a)} \sigma(\alpha')(tr) \cdot \nu_{tr}(t) & \text{if } \alpha = \alpha' a t. \end{cases}$$

Standard measure theoretical arguments ensure that $\nu_{\sigma,s}$ extends uniquely to the σ-field generated by cones. We call the measure $\nu_{\sigma,s}$ a *probabilistic execution fragment* of $\mathcal{A}$ and we say that it is generated by σ from s. Given a finite execution fragment α, we define $\nu_{\sigma,s}(\alpha)$ as $\nu_{\sigma,s}(\alpha) = \nu_{\sigma,s}(C_\alpha) \cdot \sigma(\alpha)(\perp)$, where $\sigma(\alpha)(\perp)$ is the probability of choosing no transitions, i.e., of terminating the computation after α has occurred.

We say that there is a *weak combined transition* from $s \in S$ to $\nu \in \mathrm{Disc}(S)$ labelled by $a \in \Sigma$ that is induced by σ, denoted by $s \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu$, if there exists a scheduler σ such that the following holds for the induced probabilistic execution fragment $\nu_{\sigma,s}$:

1. $\nu_{\sigma,s}(\mathit{frags}^*(\mathcal{A})) = 1$;
2. for each $\alpha \in \mathit{frags}^*(\mathcal{A})$, if $\nu_{\sigma,s}(\alpha) > 0$ then $\mathit{trace}(\alpha) = \mathit{trace}(a)$;
3. for each state t, $\nu_{\sigma,s}(\{\, \alpha \in \mathit{frags}^*(\mathcal{A}) \mid \mathit{last}(\alpha) = t \,\}) = \nu(t)$.

We say that there is a *weak transition* from $s \in S$ to $\nu \in \mathrm{Disc}(S)$ labelled by $a \in \Sigma$ that is induced by σ, denoted by $s \overset{a}{\Longrightarrow} \nu$, if there exists a Dirac scheduler σ inducing $s \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu$.
We say that there is a *weak hyper transition* from $\rho \in \mathrm{Disc}(S)$ to $\nu \in \mathrm{Disc}(S)$ labelled by $a \in \Sigma$, denoted by $\rho \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu$, if there exists a family of weak combined transitions $\{s \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu_s\}_{s \in \mathrm{Supp}(\rho)}$ such that $\nu = \bigoplus_{s \in \mathrm{Supp}(\rho)} \rho(s) \cdot \nu_s$.

Given two weak hyper transitions, it is known that their concatenation is still a weak hyper transition, provided that one of the two weak hyper transitions is labelled by τ.

Lemma 1 (cf. [14, Prop. 3.6]). *Given a PA $\mathcal{A}$ and an action a, if there exist two weak hyper transitions $\nu_1 \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu_2$ and $\nu_2 \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_3$ (or $\nu_1 \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_2$ and $\nu_2 \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu_3$), then there exists the weak hyper transition $\nu_1 \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu_3$.*

In the remainder of the paper we make use of this lemma without mentioning it further. The following technical lemma allows us to decompose a weak hyper transition $\mu \overset{a}{\Longrightarrow}_{\mathrm{c}} \mu'$ into several weak hyper transitions $\mu_i \overset{a}{\Longrightarrow}_{\mathrm{c}} \mu'_i$. This can be seen as an extension of the family of weak combined transitions to a family of generic weak hyper transitions.

Lemma 2 (cf. [7, Lemmas 5 and 6]). *Let $\mu, \mu' \in \mathrm{Disc}(S)$ and $k \in \mathbb{N}$. $\mu \overset{a}{\Longrightarrow}_{\mathrm{c}} \mu'$ iff $\mu = \mu_1 \oplus \cdots \oplus \mu_k$ for subdistributions $\mu_1, \dots, \mu_k$ and for each $i = 1, \dots, k$ a distribution μ'_i exists, such that $\mu_i \overset{a}{\Longrightarrow}_{\mathrm{c}} \mu'_i$ and $\mu' = \bigoplus_{i=1,\dots,k} \mu'_i$.*

We will often lift mappings defined on a set of states S to mappings over distributions $\mathrm{Disc}(S)$ in a generic way.

Definition 1 (Lifting of Functions). *Given arbitrary sets S and M, and $\mu \in \mathrm{Disc}(S)$, we* lift *a mapping $b\colon S \to M$ to $b\colon \mathrm{Disc}(S) \to \mathrm{Disc}(M)$ by defining $(b(\mu))(m) = \sum_{s \in b^{-1}(m)} \mu(s)$ for each $m \in M$.*

3 Bisimulations

In the following, we define strong and weak (probabilistic) bisimulations. Let $\rightsquigarrow \in \{\longrightarrow, \longrightarrow_{\mathrm{c}}, \Longrightarrow, \Longrightarrow_{\mathrm{c}}\}$.

Definition 2 (Generic Bisimulation). *Let $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$ be a PA. An equivalence relation $\mathcal{R} \subseteq S \times S$ is a $\rightsquigarrow$-bisimulation if for every action $a \in \Sigma$, distribution $\mu \in \mathrm{Disc}(S)$, and states $s, s' \in S$, with $s \; \mathcal{R} \; s'$ it holds that $s \xrightarrow{a} \mu$ implies $s' \overset{a}{\rightsquigarrow} \gamma$ for some γ and $\mu \; \mathcal{L}(\mathcal{R}) \; \gamma$.*

We denote by $\asymp_{\rightsquigarrow}$ the union of all $\rightsquigarrow$-bisimulations. Two *PA*s $\mathcal{A}$, $\mathcal{A}'$ are $\rightsquigarrow$-bisimilar, written $\mathcal{A} \asymp_{\rightsquigarrow} \mathcal{A}'$ if their initial states are bisimilar in the direct sum of the two automata. We recover the standard characterization for strong and weak bisimilarities from this definition as follows:

1. Strong Bisimilarity for *LTS*, denoted $\sim_S$, is $\asymp_{\longrightarrow}$.
2. Strong Probabilistic Bisimilarity for *PA*, denoted $\sim$, is $\asymp_{\longrightarrow_c}$.
3. Weak Bisimilarity for *LTS*, denoted $\approx_S$, is $\asymp_{\Longrightarrow}$.
4. Weak Probabilistic Bisimilarity for *PA*, denoted $\approx$, is $\asymp_{\Longrightarrow_c}$.

For the rest of the paper, we let the symbol $\asymp$ range over $\{\sim, \sim_S, \approx, \approx_S\}$. The relations $\sim_S$ and $\approx_S$ coincide with the respective notions of strong and weak bisimilarity on *LTS* [15]. The same holds for the probabilistic bisimilarities $\sim$ and $\approx$ on *PA*s [18]. In the sequel we assume that bisimilarities are only applied to suitable automata, for example, if we write $\mathcal{A} \sim_S \mathcal{A}'$, we implicitly assume $\mathcal{A}, \mathcal{A}' \in$ *LTS*.

4 Preorders

The size of an automaton is usually expressed in terms of the size of the set of states $|S|$ and the size of the transition relation $|\mathcal{T}|$ of the automaton. The complexity of algorithms working on probabilistic automata often depends exactly on those two metrics. A less commonly considered metric is the number of target states of a transition reached with a probability greater than zero. Especially in practical applications it is known that the first two of these metrics – the number of states and transitions of an automaton – can be drastically reduced while preserving its behaviour wrt. some notion of bisimilarity. In contrast, the last metric is usually considered a constant, but in some cases it can be reduced as well. We will formalize these three metrics by means of three preorder relations, thus allowing us to define the notion of *minimal automata* up to bisimilarity.

To capture the last of the three metrics, we introduce the following definition.

Definition 3 (Transition Fanout). *For a distribution $\mu \in Dist(S)$ we define $\|\mu\| = |Supp(\mu)|$. For a set of transitions T we define $\|T\| = \sum_{(s,a,\mu) \in T} \|\mu\|$.*

Definition 4 (Size Preorders). *Let $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$ and $\mathcal{A}' = (S', \bar{s}', \Sigma', \mathcal{T}')$ be two PAs, and let $\asymp$ be a notion of bisimilarity. We write*

- *$\mathcal{A} \underset{\frown}{\precsim}^{|S|} \mathcal{A}'$ if $\mathcal{A} \asymp \mathcal{A}'$ and $|S| \leq |S'|$,*
- *$\mathcal{A} \underset{\frown}{\precsim}^{|\mathcal{T}|} \mathcal{A}'$ if $\mathcal{A} \asymp \mathcal{A}'$ and $|\mathcal{T}| \leq |\mathcal{T}'|$, and*
- *$\mathcal{A} \underset{\frown}{\precsim}^{\|\mathcal{T}\|} \mathcal{A}'$ if $\mathcal{A} \asymp \mathcal{A}'$ and $\|\mathcal{T}\| \leq \|\mathcal{T}'\|$.*

We let from now on $\preceq$ range over $\underset{\frown}{\precsim}^{|S|}$, $\underset{\frown}{\precsim}^{|\mathcal{T}|}$, and $\underset{\frown}{\precsim}^{\|\mathcal{T}\|}$ for $\asymp \in \{\sim, \sim_S, \approx, \approx_S\}$, if not mentioned otherwise. It is easy to verify that these relations are preorders.

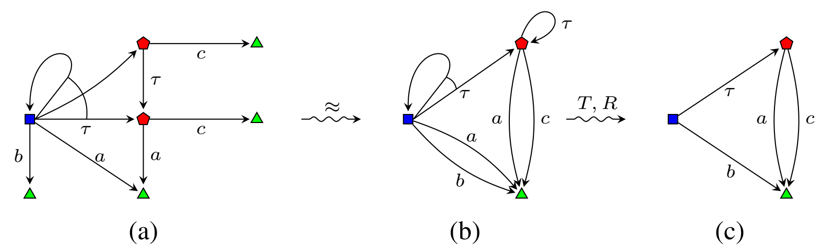


Fig. 1. (a) Example PA, (b) Quotient reduction. (c) Rescaling of convex-transitive reduction.

Definition 5 **($\preceq$-Minimal Automata).** *We call a PA $\mathcal{A}$* $\preceq$-minimal*, if whenever $\mathcal{A}' \preceq \mathcal{A}$ for some PA $\mathcal{A}'$, then also $\mathcal{A} \preceq \mathcal{A}'$.*

Lemma 3 **(Existence of $\preceq$-Minimal Automata).** *For every PA $\mathcal{A}$ there exists a PA $\mathcal{A}'$ such that $\mathcal{A}' \asymp \mathcal{A}$ and $\mathcal{A}'$ is $\preceq$-minimal.*

For each of the preorders considered, the proof of this lemma exploits that for every automaton the respective metric is a finite natural number and at least 0.

As each relation $\preceq$ is a preorder, minimal automata are not necessarily unique. For example, two $\underset{\frown}{\preceq}^{|S|}$-minimal automata $\mathcal{A}$ and $\mathcal{A}'$ with $\mathcal{A} \asymp \mathcal{A}'$ may differ in the underlying set of states, and/or transitions. This will be investigated in Section 6.

5 Reductions

In this section, we introduce and formalize several methods to reduce the size of an automaton. Except for the first method, quotient reduction, the methods are especially tailored towards one or two distinct notions of bisimilarity. We summarize the properties of the reductions at the end of this section. We will further show that each reduction can be computed in polynomial time.

5.1 Quotient Reduction

Definition 6 **(Quotient Automaton).** *Let $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$ be a PA and $\mathcal{P}(S) = \{ C \mid C \subseteq S \}$. Given an equivalence relation $\asymp$ on S, we define the* quotient PA $[\mathcal{A}]_{\asymp}$ with respect to $\asymp$ *as the reachable fragment of the PA $(S/_{\asymp}, [\bar{s}]_{\asymp}, \Sigma, [\mathcal{T}]_{\asymp})$ where (i) the equivalence class mapping $[\,\cdot\,]_{\asymp} : S \to \mathcal{P}(S)$ is defined for every $s \in S$ as $[s]_{\asymp} = \{ s' \mid s' \asymp s \}$, (ii) $S/_{\asymp} = \{ [s]_{\asymp} \mid s \in S \}$, and (iii) $[\mathcal{T}]_{\asymp} = \{ ([s]_{\asymp}, a, [\mu]_{\asymp}) \mid (s, a, \mu) \in \mathcal{T} \}$.*

Note that $[\mu]_{\asymp}$ means lifting the quotient mapping on states $[\cdot]_{\asymp}$ to distributions according to Definition 1.

Definition 7 **(Quotient Reduction).** *We write $\mathcal{A} \overset{\asymp}{\rightsquigarrow} \mathcal{A}'$ if $\mathcal{A}' = [\mathcal{A}]_{\asymp}$.*

Fig. 1(b) shows the result of applying Def. 7 to weak bisimilarity and the *PA* in Fig. 1(a).

5.2 Convex Reduction

In essence, strong probabilistic bisimilarity $\sim$ enhances standard bisimilarity by the idea that the observable behaviour of a system is closed under convex combinations of transitions. Using this fact, we minimize the number of transitions in a *PA* by replacing the transitions of each state by a unique and *minimal* set of generating transitions.

Definition 8. *Let $P = \{p_1, \ldots, p_n \in \mathbb{R}^k\}$ be a finite set of points in $\mathbb{R}^k$. We call $\mathit{CHull}(P) = \{\, c \in \mathbb{R}^k \mid \exists c_1, \ldots, c_n \geq 0 : \sum_{i=1}^{n} c_i = 1$ and $c = \sum_{i=1}^{n} c_i \cdot p_i \,\}$ the convex hull of P.*

C is a *finitely generated convex set*, if $C = \mathit{CHull}(P)$ for a finite set $P \subseteq \mathbb{R}^k$. The following lemma guarantees the optimality of our approach with respect to $\precsim^{|\mathcal{T}|}$.

Lemma 4 (cf. [3, Sec. 2]). *Every finitely generated convex set C has a unique minimal set of generators $\mathit{Gen}(C)$ such that $C = \mathit{CHull}(\mathit{Gen}(C))$.*

Definition 9 (Convex Reduction). *Let $\mathcal{A}$ be a PA. We write $\mathcal{A} \overset{C}{\rightsquigarrow} \mathcal{A}'$ if the automaton $\mathcal{A}'$ differs from $\mathcal{A}$ only by replacing the set $\mathcal{T}$ by the set $\mathcal{T}'$, where*

$$(s, a, \gamma) \in \mathcal{T}' \text{ if and only if } \gamma \in \mathit{Gen}(\mathit{CHull}(\{\, \mu \mid (s, a, \mu) \in \mathcal{T} \,\})).$$

5.3 Convex-Transitive Reduction

Just like strong probabilistic bisimilarity, weak probabilistic bisimilarity embodies the idea that the observable behaviour of a system is closed under convex combinations. Yet, this has to be interpreted for weak transitions. Finding a minimal set of generators turns out to be harder in this setting, as the behaviour of each state s no longer only depends on (convex combinations of) single step transitions leaving s. Instead, reachable distributions are now characterized by arbitrarily complex schedulers and their convex combinations. This convex set is known to be finitely generated [3].

We take inspiration from the standard approach followed in transitive reduction of order relations. Intuitively, this is the opposite of the transitive closure operation, and is achieved by removing transitions that can be reconstructed from other transitions by transitivity. In this spirit, we propose a simple algorithm that iteratively removes transitions, as long as their target distribution can also be reached by a weak combination of other transitions. Similar to transitive reduction on order relations, this reduction algorithm has polynomial complexity.

We will later show that this reduction leads to a minimal result with respect to $\precapprox^{|\mathcal{T}|}$, if applied on a model that a priori has been subjected to a quotient reduction.

Definition 10 (Convex-Transition Reduction Preorder).
Given the PAs $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$ and $\mathcal{A}' = (S', \bar{s}', \Sigma', \mathcal{T}')$, we write $\mathcal{A} \subseteq_{\mathcal{T}} \mathcal{A}'$ if and only if $\mathcal{T} \subseteq \mathcal{T}'$, $S = S'$, $\Sigma = \Sigma'$, $\bar{s} = \bar{s}'$, and for each transition $(s, a, \mu) \in \mathcal{T}'$ there exists a weak combined transition $s \xRightarrow{a}_{\mathrm{c}} \mu$ in $\mathcal{A}$.

Definition 11 ($\subseteq_{\mathcal{T}}$-Minimal Automata). *We call a PA $\mathcal{A}$* $\subseteq_{\mathcal{T}}$-minimal, *if whenever $\mathcal{A}' \subseteq_{\mathcal{T}} \mathcal{A}$ for some PA $\mathcal{A}'$, then also $\mathcal{A} \subseteq_{\mathcal{T}} \mathcal{A}'$.*

Lemma 5 (Existence of $\subseteq_{\mathcal{T}}$-Minimal Automata). *For every PA $\mathcal{A}$ there exists a PA $\mathcal{A}'$ such that $\mathcal{A}' \approx \mathcal{A}$ and $\mathcal{A}'$ is $\subseteq_{\mathcal{T}}$-minimal.*

Definition 12 (Convex Transitive Reduction). *Let $\mathcal{A}$ be a PA. We write $\mathcal{A} \overset{T}{\rightsquigarrow} \mathcal{A}'$ if $\mathcal{A}' \subseteq_{\mathcal{T}} \mathcal{A}$ and $\mathcal{A}'$ is $\subseteq_{\mathcal{T}}$-minimal.*

Notably, this reduction relation is non-deterministic, i.e., non-functional. But, as we will show in Section 6, it is unique up to isomorphism ($=_{iso}$), if applied to a quotient reduced automaton. The overall result will therefore be unique up to isomorphism. As a special case, this reduction can be applied to non-probabilistic transition systems (*LTS*s), where it then coincides with transitive reduction of order relations. For this it is irrelevant that this reduction allows to combine transitions, as long as we work on a quotient reduced system, because in that system bisimilar states have been collapsed into a single representative. Thus, a Dirac transition to a single state can only be matched by a Dirac transition to precisely that state. In the *LTS* setting, $\overset{T}{\rightsquigarrow}$ preserves $\approx_S$, and in fact is a necessary step to arrive at the transition minimal quotient. As a side note, though this must have been considered in the context of tools exploiting weak bisimilarity [5, 8], we are not aware of a publication mentioning this point.

5.4 Rescaling

A particular fine point of weak probabilistic bisimilarities [1] is related to internal transitions that induce a nonzero chance of residing inside the class. If looking at the quotient, this concerns any internal transition (s, τ, μ) that contains the source state s with nontrivial probability, i.e., $0 < \mu(s) < 1$. For those transitions, we can renormalise the probability of all other states in the support set by $1 - \mu(s)$ without breaking weak bisimilarity. In other words, such μ can be replaced by the rescaled distribution $\mu \backslash s$.

Definition 13 (Rescaling). *Let $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$ be a PA. We write $\mathcal{A} \overset{R}{\rightsquigarrow} \mathcal{A}'$ if $\mathcal{A}' = (S, \bar{s}, \Sigma, \mathcal{T}')$ such that for each $(s, a, \mu') \in \mathcal{T}'$, either $a \in E$ and $(s, a, \mu') \in \mathcal{T}$, or $a \in H$ and there exists $(s, \tau, \mu) \in \mathcal{T}$ such that $\mu(s) < 1$ and $\mu' = \mu \backslash s$.*

As it will turn out, this reduction is the final step to obtain minimality with respect to $\precsim^{\|\mathcal{T}\|}_{\approx}$ if applied a posteriori to the other two reductions, $\overset{\approx}{\rightsquigarrow}$ and $\overset{T}{\rightsquigarrow}$. Figure 1(c) depicts the result of applying this sequence of reductions on the *PA* in Figure 1(a). Figure 1(b) shows the automaton after it has been subjected to quotient reduction only.

5.5 Properties of Reductions

We summarize preservation and computability properties of the reduction relations.

Lemma 6 (Preservation of Bisimilarities)

1. *$\mathcal{A} \overset{\asymp}{\rightsquigarrow} \mathcal{A}'$ implies $\mathcal{A} \asymp \mathcal{A}'$ for each $\mathcal{A}, \mathcal{A}'$ and $\asymp \in \{\sim, \sim_S, \approx, \approx_S\}$.*
2. *$\mathcal{A} \overset{C}{\rightsquigarrow} \mathcal{A}'$ implies $\mathcal{A} \sim \mathcal{A}'$ for each $\mathcal{A}, \mathcal{A}' \in$ PA.*
3. *$\mathcal{A} \overset{T}{\rightsquigarrow} \mathcal{A}'$ implies $\mathcal{A} \asymp \mathcal{A}'$ for each $\mathcal{A}, \mathcal{A}'$ and $\asymp \in \{\approx_S, \approx\}$.*
4. *$\mathcal{A} \overset{R}{\rightsquigarrow} \mathcal{A}'$ implies $\mathcal{A} \approx \mathcal{A}'$ for each $\mathcal{A}, \mathcal{A}' \in$ PA.*

Proof. **Proof for $\overset{\asymp}{\rightsquigarrow}$, $\overset{C}{\rightsquigarrow}$ and $\overset{T}{\rightsquigarrow}$:** The result follows almost immediately from the definitions of the reductions.
Proof for $\overset{R}{\rightsquigarrow}$: Since by definition of $\overset{R}{\rightsquigarrow}$, $\mathcal{A}$ and $\mathcal{A}'$ have the same set of states, we use ν to refer to distributions from both $\mathcal{A}$ and $\mathcal{A}'$; we still use s' to remark that we consider the state s in $\mathcal{A}'$.

Let $\mathcal{I}$ be the equivalence relation on $S \uplus S'$ whose set of classes is $\{\, \{s, s'\} \mid s \in S \,\}$, i.e., we relate each state s with its primed counterpart in $\mathcal{A}'$. $\mathcal{I}$ is a weak probabilistic bisimulation for $\mathcal{A}$ and $\mathcal{A}'$: let $s \mathrel{\mathcal{I}} t$ and $s \xrightarrow{a} \nu$; if $s = t$, then also t enables the transition $t \xrightarrow{a} \nu$ and $\nu \mathrel{\mathcal{L}(\mathcal{I})} \nu$. Suppose that $s \neq t$; if $a \in E$, then also t enables the transition $t \xrightarrow{a} \nu$, thus $\nu \mathrel{\mathcal{L}(\mathcal{I})} \nu$. Now, consider $a \in H$: if $s \in S$ and $t \in S'$, i.e., $t = s'$, then t is able to match such transition by the weak combined transition $t \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu$ as induced by the scheduler σ such that $\sigma(t)(\perp) = \nu(s)$, $\sigma(t)(tr) = 1 - \nu(s)$, and $\sigma(\alpha)(\perp) = 1$ for each finite execution fragment $\alpha \neq t$, where $tr = (t, \tau, \nu \backslash s)$. Note that this applies also when $\nu = \delta_s$ as the resulting scheduler assigns $\sigma(t)(\perp) = \nu(s) = 1$ so the induced weak combined transition is $t \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \delta_t$ and $\delta_s \mathrel{\mathcal{L}(\mathcal{I})} \delta_t$. Otherwise, if $s \in S'$ and $t \in S$, i.e., $s = t'$, then $s \xrightarrow{a} \nu$ is actually a transition $s \xrightarrow{a} \rho \backslash s$ that t is able to match by the weak combined transition $t \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu$ as induced by the determinate scheduler σ such that $\sigma(\alpha)(tr') = 1$ for each $\alpha \in \mathit{frags}^*(\mathcal{A})$ with $\mathit{last}(\alpha) = t$, and $\sigma(\alpha)(\perp) = 1$ for each finite execution fragment α with $\mathit{last}(\alpha) \neq t$ where $tr' = (t, \tau, \rho)$. □

Proposition 1 (Computability of Reductions). *For every PA $\mathcal{A}$, a PA $\mathcal{A}'$ can be found in polynomial time, such that $\mathcal{A} \rightsquigarrow \mathcal{A}'$ for $\rightsquigarrow \in \{\overset{\asymp}{\rightsquigarrow}, \overset{C}{\rightsquigarrow}, \overset{T}{\rightsquigarrow}, \overset{R}{\rightsquigarrow}\}$.*

Proof (outline). The result for $\overset{\asymp}{\rightsquigarrow}$ follows by the corresponding polynomial decision procedures [3, 8, 10, 11, 16] and reachability analysis; $\overset{C}{\rightsquigarrow}$ requires for each state and each enabled action to solve $\mathcal{O}(|\mathcal{T}|)$ linear programming problems (cf. [3, Sec. 6]) in order to find the set of generators of reachable distributions; $\overset{R}{\rightsquigarrow}$ can be obtained by computing for each transition $s \xrightarrow{\tau} \nu$ the distribution $\nu \backslash s$ that requires at most $\mathcal{O}(|S|)$ operations; finally, $\overset{T}{\rightsquigarrow}$ can be computed by iteratively refining $\mathcal{A}$ by removing one transition obtaining $\mathcal{A}'$ and deciding whether $\mathcal{A} \approx \mathcal{A}'$. Since this is polynomial [10] and the check is performed at most $|\mathcal{T}|$ times, computing $\overset{T}{\rightsquigarrow}$ is polynomial. □

6 Normal Forms

We are now concerned with minimality and uniqueness properties induced by the reduction operations with respect to the metrics discussed. To discuss uniqueness, it is convenient to introduce normal forms as means to canonically represent automata in such a way that two automata are equivalent if and only if they have identical normal forms. Or better, if and only if the normal forms are identical up to isomorphism (structural identity). Two *PA*s $\mathcal{A} = (S, \bar{s}, \Sigma, \mathcal{T})$ and $\mathcal{A}' = (S', \bar{s}', \Sigma', \mathcal{T}')$ are *isomorphic*, denoted by $\mathcal{A} =_{iso} \mathcal{A}'$, if $\Sigma = \Sigma'$ and there is a bijective mapping $b \colon S \to S'$ such that $b(\bar{s}) = \bar{s}'$ and $(s, a, \mu) \in \mathcal{T}$ if and only if $(b(s), a, b(\mu)) \in \mathcal{T}'$.

Definition 14 (Normal Form). *Given an equivalence relation $\asymp$ over PAs, we call $NF\colon PA \to PA$ a* normal form*, if it satisfies for every PA $\mathcal{A}$*

- *$NF(\mathcal{A}) \asymp \mathcal{A}$, and*
- *for every PA $\mathcal{A}'$ it holds that $\mathcal{A} \asymp \mathcal{A}'$ if and only if $NF(\mathcal{A}) =_{iso} NF(\mathcal{A}')$.*

It is natural to strive for normal forms that are distinguished in a certain sense. Not surprisingly, we will strive for normal forms that are distinguished as being the smallest possible representation of the behaviour they represent. In the following, we call a total and functional subset of a binary relation $r \subseteq PA \times PA$ a *function in* r. Note that every function in r is a mapping $PA \to PA$.

Definition 15 (Normal Form Instances)

- *Let $NF_{\sim_S} = \overset{\sim_S}{\rightsquigarrow}$.*
- *Let $NF_{\approx_S}$ be an arbitrary function in $\overset{\approx_S}{\rightsquigarrow} \circ \overset{T}{\rightsquigarrow}$.*
- *Let $NF_{\sim} = \overset{\sim}{\rightsquigarrow} \circ \overset{C}{\rightsquigarrow}$.*
- *Let $NF_{\approx}$ be an arbitrary function in $\overset{\approx}{\rightsquigarrow} \circ \overset{T}{\rightsquigarrow} \circ \overset{R}{\rightsquigarrow}$.*

Theorem 1. *Let $\asymp \in \{\sim, \sim_S, \approx, \approx_S\}$.*

1. ***Minimality:** $NF_{\asymp}(\mathcal{A})$ is $\underset{\frown}{\preceq}^{|S|}$, $\underset{\frown}{\preceq}^{|T|}$, and $\underset{\frown}{\preceq}^{\|T\|}$-minimal for every $\mathcal{A} \in PA$.*
2. ***Uniqueness of minimals:** If $\mathcal{A}$ and $\mathcal{A}'$ are $\underset{\frown}{\preceq}^{|S|}$, $\underset{\frown}{\preceq}^{|T|}$, and $\underset{\frown}{\preceq}^{\|T\|}$-minimal automata, and $\mathcal{A} \asymp \mathcal{A}'$, then also $\mathcal{A} =_{iso} \mathcal{A}'$,*
3. ***Normal forms:** $NF_{\asymp}$ is uniquely defined up to $=_{iso}$, and is a normal form.*

It is straightforward to check that all normal forms $NF_{\asymp}$ above are indeed mappings. Furthermore, by Lemma 6, it follows that in each of the cases $NF_{\asymp}(\mathcal{A}) \asymp \mathcal{A}$.

The remainder of this section is devoted to the proof of Theorem 1. We begin with a lemma that we use often.

Lemma 7 (Preservation of Minimality). *Let $\preceq \in \{\underset{\frown}{\preceq}^{|S|}, \underset{\frown}{\preceq}^{|T|}, \underset{\frown}{\preceq}^{\|T\|}, \subseteq_T\}$. If $\mathcal{A} =_{iso} \mathcal{A}'$ and $\mathcal{A}$ is $\preceq$-minimal, then $\mathcal{A}'$ is $\preceq$-minimal, too.*

For each normal form, the proof will refer to the following crucial, but already folklore insight, that the quotient automaton is minimal with respect to the number of states.

Lemma 8 (State Minimality of Quotient Automata). *For every $\mathcal{A} \in PA$, the automaton $\mathcal{A}'$ with $\mathcal{A} \overset{\asymp}{\rightsquigarrow} \mathcal{A}'$ is $\underset{\frown}{\preceq}^{|S|}$-minimal.*

Next, we show that $\underset{\frown}{\preceq}^{|S|}$and $\underset{\frown}{\preceq}^{|T|}$-minimality can be achieved at the same time in one automaton. For bisimilarities on *LTS*s, this is enough to conclude also $\underset{\frown}{\preceq}^{\|T\|}$-minimality, as this always coincides with $\underset{\frown}{\preceq}^{|T|}$-minimality here (as all transition have the form (s, a, δ_t)).

Lemma 9 (Compatibility of $\underset{\frown}{\preceq}^{|S|}$and $\underset{\frown}{\preceq}^{|T|}$-minimality). *For every PA $\mathcal{A}$ there exists a PA $\mathcal{A}'$ with $\mathcal{A}' \asymp \mathcal{A}$, which is $\underset{\frown}{\preceq}^{|S|}$and $\underset{\frown}{\preceq}^{|T|}$-minimal.*

Proof. By Lemma 3, there exists a *PA* $\mathcal{A}$ that is $\underset{\frown}{\preceq}^{|T|}$-minimal. Consider $[\mathcal{A}]_{\asymp}$. From Definition 6 it is clear that for every transition of $[\mathcal{A}]_{\asymp}$ there exists a transition in $\mathcal{A}$. Thus, $[\mathcal{A}]_{\asymp} \underset{\frown}{\preceq}^{|T|} \mathcal{A}$, and hence, $[\mathcal{A}]_{\asymp}$ must also be $\underset{\frown}{\preceq}^{|T|}$-minimal. Furthermore, by Lemma 8, $[\mathcal{A}]_{\asymp}$ must also be $\underset{\frown}{\preceq}^{|S|}$-minimal, and finally with Lemma 6 $\mathcal{A} \asymp \mathcal{A}'$ follows. □

Strong Bisimilarities

Lemma 10 (Canonicity of Normal Form). *Let $\asymp \in \{\sim_S, \sim\}$, $\mathcal{A} \in$ PA, and $\mathcal{A}' = NF_{\asymp}(\mathcal{A})$. For every $\underset{\frown}{\preceq}^{|S|}$and $\underset{\frown}{\preceq}^{|\mathcal{T}|}$-minimal PA $\mathcal{A}_m$ with $\mathcal{A}_m \asymp \mathcal{A}$, also $\mathcal{A}_m =_{iso} \mathcal{A}'$.*

Proof. We skip the proof for $\asymp\, = \,\sim_S$ and proceed with the more complicated case of $\asymp\, = \,\sim$. Let $\mathcal{A}_m$ be a $\underset{\frown}{\preceq}^{|S|}$and $\underset{\frown}{\preceq}^{|\mathcal{T}|}$-minimal automaton. Recall that $NF_{\sim} = \overset{\sim}{\rightsquigarrow} \circ \overset{C}{\rightsquigarrow}$. As applying $\overset{\sim}{\rightsquigarrow}$ to $\mathcal{A}$ leads to a $\precsim^{|S|}$-minimal automaton according to Lemma 8, and $\overset{C}{\rightsquigarrow}$ obviously does not alter the number of states, $NF_{\sim}(\mathcal{A}) = \mathcal{A}'$ is $\precsim^{|S|}$-minimal, and thus $|S_m| = |S'|$, as $\mathcal{A}_m$ is $\precsim^{|S|}$-minimal by assumption.

Since $\mathcal{A}' \sim \mathcal{A}$ and $\mathcal{A} \sim \mathcal{A}_m$, we have $\mathcal{A}' \sim \mathcal{A}_m$. We will now argue that $b = \sim \cap\, (S' \times S_m)$ is in fact a suitable mapping to establish $\mathcal{A}' =_{iso} \mathcal{A}_m$. We start by showing that b is functional, injective and surjective. Assume b is not injective. Then there must exist states $s_1, s_2 \in S'$ and $t \in S_m$, such that $b(s_1) = t$ and $b(s_2) = t$. But this implies $s_1 \sim t$ and $s_2 \sim t$. By transitivity, this implies $s_1 \sim s_2$, contradicting Lemma 8. Functionality can be shown similarly. We skip the details. If b is not surjective, this would immediately contradict the assumption that $\mathcal{A}_m$ is $\precsim^{|S|}$-minimal, since then any state $t \in \mathcal{A}_m$ for which no $s \in S'$ exists, such that $b(s) = t$ could be removed without violating $\mathcal{A}' \sim \mathcal{A}_m$.

Most of the other conditions that have to be checked to show that b is an isomorphism are straightforward, except for the condition

$$(s, a, \mu) \in \mathcal{T} \quad \text{if and only if} \quad (b(s), a, b(\mu)) \in \mathcal{T}'. \tag{$\star$}$$

The set of combined transitions any state s of $\mathcal{A}'$ can do must equal the set of combined transitions that $b(s)$ can do as $s \sim b(s)$. By reduction $\overset{C}{\rightsquigarrow}$, the set of transitions leaving s must be minimal, according to Lemma 4, and must also be unique. As the transitions of $b(s)$ are minimal by assumption, the uniqueness of the minimal set of generators guarantees Condition ($\star$). □

For $\sim_S$ and $\sim$, Theorem 1 now follows almost immediately by Lemma 9, Lemma 10 and Lemma 6. For $\sim_S$, we in addition need the observation that $\mathcal{A}$ is $\underset{\frown}{\preceq}^{\|\mathcal{T}\|}$-minimal if and only if it is $\underset{\frown}{\preceq}^{|\mathcal{T}|}$-minimal, as we remarked before Lemma 9. For $\sim$, the same observation holds, but follows from the uniqueness of the minimal set of generators (Lemma 4).

Weak Bisimilarities The following two lemmas are the weak counterparts to Lemma 10.

Lemma 11. *Let $\mathcal{A}$ be a PA and $\mathcal{A}' = NF_{\approx_S}(\mathcal{A})$. Let $\mathcal{A}_m$ be a $\precapprox_S^{|S|}$and $\precapprox_S^{|\mathcal{T}|}$-minimal PA satisfying $\mathcal{A}_m \approx_S \mathcal{A}$. Then $\mathcal{A}' =_{iso} \mathcal{A}_m$.*

We skip the proof of this lemma, as it is similar to, but simpler than the proof of the following lemma. Theorem 1 can then be proven in complete analogy to the proof for $\sim_S$.

It is instructive to note that in the following lemma, we need to apply the reduction $\overset{R}{\rightsquigarrow}$ to arrive at an uniqueness result. Only applying $\overset{\approx}{\rightsquigarrow}$ followed by $\overset{T}{\rightsquigarrow}$ will still lead to $\precsim^{|S|}$and $\precsim^{|\mathcal{T}|}$-minimal automata, but they will not agree up to $=_{iso}$, in full generality. Different to Lemmas 11 and 10, the following lemma is slightly more general.

Lemma 12. *Let $\mathcal{A}$ be a $\precapprox^{|S|}$-minimal PA, $\mathcal{A} \overset{T}{\rightsquigarrow} \circ \overset{R}{\rightsquigarrow} \mathcal{A}'$, and $\mathcal{A}'_m$ be a $\precapprox^{|S|}$and $\precapprox^{|\mathcal{T}|}$-minimal PA satisfying $\mathcal{A}'_m \approx \mathcal{A}$. Now let $\mathcal{A}'_m \overset{R}{\rightsquigarrow} \mathcal{A}_m$ for some $\mathcal{A}_m$. Then $\mathcal{A}' =_{iso} \mathcal{A}_m$.*

Proof. Let $\mathcal{A}_m$ and $\mathcal{A}'$ be chosen as in the claim. We then proceed similarly as in the proof of Lemma 10 to show that $b = {\approx} \cap (S_m \times S')$ is a bijection. Then we will be able to establish that b is a suitable mapping to establish $\mathcal{A}_m =_{iso} \mathcal{A}'$.

Assume, to derive a contradiction, that b is not an isomorphism. Since b is a bijection between S_m and S' (note that all automata in this lemma must be $\precsim^{|S|}$-minimal), in order to have $\mathcal{A}_m \neq_{iso} \mathcal{A}'$ there must exist $s \in S_m$, $t \in S'$ with $s \approx t$ (i.e., $b(s) = t$), and (i) either a transition $s \xrightarrow{a} \nu_s \in \mathcal{T}_m$ but there does not exist $t \xrightarrow{a} \nu_t \in \mathcal{T}'$ such that $\nu_s \; \mathcal{L}(\approx) \; \nu_t$, i.e., there does not exist a transition $t \xrightarrow{a} \nu_t \in \mathcal{T}'$ such that $\nu_t = b(\nu_s)$, or (ii) a transition $t \xrightarrow{a} \nu_t \in \mathcal{T}'$ but there does not exist $s \xrightarrow{a} \nu_s \in \mathcal{T}_m$ such that $\nu_s \; \mathcal{L}(\approx) \; \nu_t$. We proceed with the proof of (i).

Note that this cannot be caused by two transitions with $\nu_t \neq b(\nu_s)$ but $b(\nu_s \backslash s) = \nu_t \backslash t$, since both automata are rescaled. However, since $s \approx t$, it follows that there exists $t \overset{a}{\Longrightarrow}_{\mathrm{c}} \nu_t$ such that $\nu_s \; \mathcal{L}(\approx) \; \nu_t$. Now, there are two cases: either $a \in E$, or $a \in H$. We provide the detailed proof for $a = \tau$ whose schematic proof idea is depicted below; the case $a \neq \tau$ is similar.

$$\begin{tikzcd}
s \arrow[r, Rightarrow, "\tau", "\mathrm{c}"'] \arrow[ddd, "\tau"'] & \gamma_s^i \arrow[l, Rightarrow, bend left, "\tau", "\mathrm{c}"'] \arrow[d, Rightarrow, "\tau", "\mathrm{c}"'] & & & \\
& \nu'_s & & \gamma_t^1 & \\
& \oplus \; = \; \nu_s \; \mathcal{L}(\approx) \; \nu_t & & \oplus \; \vdots \; \oplus \arrow[ll, Rightarrow, "\tau"', "\mathrm{c}"] & t \arrow[ul, "\tau"'] \arrow[dl, "\tau"] \\
\nu_s \arrow[r, Rightarrow, "\tau", "\mathrm{c}"'] & \rho_s & & \gamma_t^n &
\end{tikzcd}$$

Let σ_t be the scheduler inducing $t \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_t$ and $t \xrightarrow{\tau} \gamma_t^1, \ldots, t \xrightarrow{\tau} \gamma_t^n$ be all transitions such that $\sigma_t(t)(t \xrightarrow{\tau} \gamma_t^i) > 0$ and $\gamma_t^i \not\mathcal{L}(\approx) \; \nu_s$, that is, $t \xrightarrow{\tau} \gamma_t^i$ is a transition used in the first step of the weak combined transition $t \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_t$; it is immediate to see that $(\bigoplus_{i=1}^n \gamma_t^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_t$. Since $s \approx t$, it follows that there exists γ_s^i for each $1 \leq i \leq n$ such that $s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \gamma_s^i$ and $\gamma_s^i \; \mathcal{L}(\approx) \; \gamma_t^i$. Furthermore, $(\bigoplus_{i=1}^n \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_s$, as $(\bigoplus_{i=1}^n \gamma_t^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_t$ and $\nu_t = b(\nu_s)$.

Now, consider a generic γ_s^j; there are two cases depending on whether $s \xrightarrow{\tau} \nu_s$ is used to reach ν_s. If it is not used by any of the γ_s^i, then there exists the weak combined transition $s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} (\bigoplus_{i=1}^n \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_s$ that does not involve $s \xrightarrow{\tau} \nu_s$, hence $s \xrightarrow{\tau} \nu_s$ can be omitted. This contradicts the $\precapprox^{|\mathcal{T}|}$-minimality of $\mathcal{A}_m$.

So, suppose that $s \xrightarrow{\tau} \nu_s$ is used in order to reach ν_s. Since $(\bigoplus_{i=1}^n \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_s$, we may split this hyper-transition into two parts according to Lemma 2, depending on whether $s \xrightarrow{\tau} \nu_s$ is chosen by the scheduler with non-zero probability: $(\bigoplus_{i=1}^n \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu'_s$ with weight $c_1 \geq 0$ that does not involve $s \xrightarrow{\tau} \nu_s$, and $(\bigoplus_{i=1}^n \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \delta_s$ with weight $c_2 > 0$ that involves $s \xrightarrow{\tau} \nu_s$ such that $c_1 + c_2 = 1$ and there exists ρ_s

such that ($s \xrightarrow{\tau} \nu_s$ and) $\nu_s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \rho_s$ and $\nu_s = (c_1\nu'_s \oplus c_2\rho_s)$. Note that we use ρ_s instead of ν_s since it may be that, in order to reach distribution equivalent to ν_s, we have to adjust probabilities by performing more steps. Now, consider the convex combination of the two weak combined transitions $Tr_1 = s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} (\bigoplus_{i=1}^{n} \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu'_s$ and $Tr_2 = s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} (\bigoplus_{i=1}^{n} \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \delta_s \xrightarrow{\tau} \nu_s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \rho_s$, with weights c_1 and c_2, respectively. Since $(c_1\nu'_s \oplus c_2\rho_s) = \nu_s$, we have that such convex combination corresponds to the weak transition $s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_s$, so we can replace the transition $s \xrightarrow{\tau} \nu_s$ by the weak combined transition $Tr = c_1 \cdot Tr_1 \oplus c_2 \cdot Tr_2$ with $\nu_s = c_1\nu'_s \oplus c_2\rho_s$. Since $s \xrightarrow{\tau} \nu_s$ still occurs in $Tr_2 = s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \delta_s \xrightarrow{\tau} \nu_s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \rho_s$, we can recursively replace it by the same weak combined transition Tr, hence, after k replacements, we have that $\nu_s = c_1\nu'_s \oplus c_2c_1\nu'_s \oplus c_2^2c_1\nu'_s \oplus \cdots \oplus c_2^k\rho_s = (\bigoplus_{l=0}^{k-1} c_1c_2^l\nu'_s) \oplus c_2^k\rho_s$, that is, $(\bigoplus_{l=0}^{k-1}(1-c_2)c_2^l\nu'_s) \oplus c_2^k\rho_s$. If we tend k to infinite, since $c_2 < 1$, we derive that $\nu_s = \nu'_s$, therefore there exists the weak combined transition $s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} (\bigoplus_{i=1}^{n} \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \nu_s$ that does not involve $s \xrightarrow{\tau} \nu_s$, hence again $s \xrightarrow{\tau} \nu_s$ can be omitted. This contradicts the $\precapprox^{|T|}$-minimality of $\mathcal{A}_m$. The proof of case (ii) is completely analogous, except that the contradictions will be derived with respect to $\subseteq_{\mathcal{T}}$, which is a result of the fact that $\mathcal{A}'$ has been reduced according to $\overset{T}{\leadsto}$.

As final note, consider the weight c_2 and suppose that $c_2 = 1$. Since $s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} (\bigoplus_{i=1}^{n} \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \delta_s$ with $(\bigoplus_{i=1}^{n} \gamma_s^i) \not\mathcal{L}(\approx)\ \delta_s$, it follows that each state in the support of $\bigoplus_{i=1}^{n} \gamma_s^i$ is actually weak bisimilar to s as the states touched in the loop $s \overset{\tau}{\Longrightarrow}_{\mathrm{c}} (\bigoplus_{i=1}^{n} \gamma_s^i) \overset{\tau}{\Longrightarrow}_{\mathrm{c}} \delta_s$ form a strongly connected component. But this contradicts the $\precapprox^{|S|}$-minimality of $\mathcal{A}_m$.

□

Corollary 1. *Let $\mathcal{A}$ be a $\precapprox^{|S|}$-minimal PA.*
$\mathcal{A}$ is $\subseteq_{\mathcal{T}}$-minimal if and only if it is $\precapprox^{|T|}$-minimal.

Proof. Let $\mathcal{A}$ be $\precapprox^{|S|}$-minimal. For the first direction of the *if and only if*, note first that by Lemma 9, a *PA* $\mathcal{A}'_m$ must exist, which is minimal with respect to $\precapprox^{|T|}$and $\precapprox^{|S|}$. Let $\mathcal{A}'_m \overset{R}{\leadsto} \mathcal{A}_m$. Clearly, $\mathcal{A}_m$ must be $\precapprox^{|S|}$and $\precapprox^{|T|}$-minimal, too. As by assumption, $\mathcal{A}$ is $\subseteq_{\mathcal{T}}$-minimal, $\mathcal{A} \overset{T}{\leadsto} \mathcal{A}$. Let $\mathcal{A}'$ satisfy $\mathcal{A} \overset{R}{\leadsto} \mathcal{A}'$. We combine the two reductions and see that $\mathcal{A} \overset{T}{\leadsto} \circ \overset{R}{\leadsto} \mathcal{A}'$. This allows us to apply Lemma 12 to obtain $\mathcal{A}' =_{iso} \mathcal{A}_m$. As $\mathcal{A}' =_{iso} \mathcal{A}_m$ implies that both have the same number of transitions, also $\mathcal{A}'$ must be $\precapprox^{|T|}$-minimal. If we can now show that also $\mathcal{A}$ and $\mathcal{A}'$ have the same number of transitions, we are done. Assume the contrary to arrive at a contradiction. As $\mathcal{A} \overset{R}{\leadsto} \mathcal{A}'$, this is only possible if there are two transitions (s, τ, μ) and (s, τ, γ) in $\mathcal{A}$ such that $\mu\backslash s = \gamma\backslash s$. But then, one of them could have been removed without changing the combined weak transitions s can perform, contradicting the assumption that $\mathcal{A}$ is $\subseteq_{\mathcal{T}}$-minimal.

For the other direction, assume $\mathcal{A}$ is in addition $\precapprox^{|T|}$-minimal. As removing transitions from $\mathcal{A}$ would lead to an automaton that is smaller with respect to $\precapprox^{|T|}$, it must be the case that any such automaton $\mathcal{A}'$ does not satisfy $\mathcal{A}' \approx \mathcal{A}$, otherwise contradicting the assumption that $\mathcal{A}$ was $\precapprox^{|T|}$-minimal. But then it immediately follows that $\mathcal{A}$ is also $\subseteq_{\mathcal{T}}$-minimal. □

Lemma 13. *If $\mathcal{A}$ is $\precapprox^{\|\mathcal{T}\|}$-minimal, then there also exists $\mathcal{A}'$, such that $\mathcal{A} \approx \mathcal{A}'$ and $\mathcal{A}'$ is $\precapprox^{|S|}$, $\precapprox^{|\mathcal{T}|}$, and $\precapprox^{\|\mathcal{T}\|}$-minimal.*

Proof. We first show that for every $\precapprox^{\|\mathcal{T}\|}$-minimal automaton $\mathcal{A}$ there is one that is also $\precapprox^{|S|}$-minimal. As candidate, we take the unique automaton $\mathcal{A}'$ such that $\mathcal{A} \overset{\approx}{\rightsquigarrow} \mathcal{A}'$. From Definitions 6 and 7 it is clear that the transitions of $\mathcal{A}'$ can be surjectively mapped to transitions of $\mathcal{A}$, such that every transition of $\mathcal{A}'$ is smaller or equal with respect to $\|\cdot\|$ than its image transition in $\mathcal{A}$. Thus, minimality with respect to $\precapprox^{\|\mathcal{T}\|}$is preserved.

Now we show that any $\mathcal{A}''$, which satisfies $\mathcal{A}' \overset{T}{\rightsquigarrow} \mathcal{A}''$ is in addition $\precapprox^{|\mathcal{T}|}$-minimal. Clearly, the numbers of states of $\mathcal{A}'$ and $\mathcal{A}''$ are the same. Furthermore, the transitions of $\mathcal{A}''$ form a subset of the transitions of $\mathcal{A}'$. Thus, as $\mathcal{A}'$ is $\precapprox^{\|\mathcal{T}\|}$-minimal, also $\mathcal{A}''$ must be $\precapprox^{\|\mathcal{T}\|}$-minimal. By Definition 12, $\mathcal{A}''$ is minimal with respect to $\subseteq_{\mathcal{T}}$, and thus, by Corollary 1, also with respect to $\precapprox^{|\mathcal{T}|}$. □

Corollary 2. *For every PA $\mathcal{A}$ there exists a PA $\mathcal{A}'$ with $\mathcal{A}' \approx \mathcal{A}$, which is $\precapprox^{|S|}$, $\precapprox^{|\mathcal{T}|}$and $\precapprox^{\|\mathcal{T}\|}$-minimal.*

Proof. Follows immediately from Lemma 3 and Lemma 13. □

Lemma 14 (Canonicity of Normal Form). *Let $\mathcal{A}' = NF_{\approx}(\mathcal{A})$. Let $\mathcal{A}_m$ be a $\precapprox^{|S|}$, $\precapprox^{|\mathcal{T}|}$, and $\precapprox^{\|\mathcal{T}\|}$-minimal automaton satisfying $\mathcal{A}_m \approx \mathcal{A}$. Then $\mathcal{A}' =_{iso} \mathcal{A}_m$.*

Proof. By Corollary 2 we know that $\mathcal{A}_m$ exists such that $\mathcal{A}_m \approx \mathcal{A}$ and $\mathcal{A}_m$ is $\precapprox^{|S|}$, $\precapprox^{|\mathcal{T}|}$ and $\precapprox^{\|\mathcal{T}\|}$-minimal. Furthermore, as $\mathcal{A}_m$ is $\precapprox^{\|\mathcal{T}\|}$-minimal, it must hold $\mathcal{A}_m \overset{R}{\rightsquigarrow} \mathcal{A}_m$. Finally, as $\mathcal{A}' = NF_{\approx}(\mathcal{A})$, there must exist $\mathcal{A}''$ such that $\mathcal{A} \overset{\approx}{\rightsquigarrow} \mathcal{A}''$ and $\mathcal{A}'' \overset{T}{\rightsquigarrow} \circ \overset{R}{\rightsquigarrow} \mathcal{A}'$, and by the Definition of $\overset{\approx}{\rightsquigarrow}$ and Lemma 8, $\mathcal{A}''$ is $\precapprox^{|S|}$-minimal. Thus, we may apply Lemma 12 to obtain our result. □

Theorem 1 now follows for $\approx$ with Corollary 2 and Lemma 14.

7 Conclusion

This paper has successfully answered the question how to compute the minimal, canonical representation of probabilistic automata under strong and weak bisimilarity, together with polynomial time minimization algorithms. Canonical forms have also appeared in axiomatic treatments of probabilistic calculi [6], but are obtained by adding transitions via saturation, so without aiming for minimality. Figure 2 summarizes what steps are needed to perform the minimization in labelled transition systems (left) and probabilistic automata (right). The triplets indicate minimality (✓) or non-minimality (×) with respect to

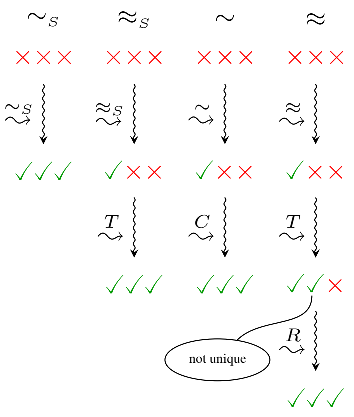

$\sim_S$
$\approx_S$
$\sim$
$\approx$
××× ××× ××× ×××
$\overset{\sim_S}{\rightsquigarrow}$
$\overset{\approx_S}{\rightsquigarrow}$
$\overset{\sim}{\rightsquigarrow}$
$\overset{\approx}{\rightsquigarrow}$
✓✓✓ ✓×× ✓×× ✓××
$\overset{T}{\rightsquigarrow}$
$\overset{C}{\rightsquigarrow}$
$\overset{T}{\rightsquigarrow}$
✓✓✓ ✓✓✓ ✓✓×
not unique
$\overset{R}{\rightsquigarrow}$
✓✓✓

Fig. 2. Algorithmic steps in minimal quotient computation

$|S|$, then $|\mathcal{T}|$, then $\|\mathcal{T}\|$. For example, ✓✓× indicates that state and transition numbers are minimal, but transition fanout size can be non-minimal.

The algorithms we developed can be exploited in an effective compositional minimization strategy for *PA*s (or *MDP*s), because strong and weak bisimilarity are congruence relations for the standard process algebraic operators. With this, we see a rich spectrum of potential applications in operations research, automated planning, and in the decision support context.

Acknowledgements. This work is supported by DFG/NWO bilateral research programme ROCKS, by the DFG as part of the SFB/TR 14 AVACS, by the EU FP7 Programme under grant agreement no. 295261 (MEALS), 318490 (SENSATION), and TREsPASS (318003). Andrea Turrini is supported by the Cluster of Excellence “Multimodal Computing and Interaction” (MMCI), part of the German Excellence Initiative. Lijun Zhang is supported by IDEA4CPS and MT-LAB, a VKR Centre of Excellence.

References

1. Baier, C., Hermanns, H.: Weak Bisimulation for Fully Probabilistic Processes. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 119–130. Springer, Heidelberg (1997)
2. Barbot, B., Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Efficient CTMC Model Checking of Linear Real-Time Objectives. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 128–142. Springer, Heidelberg (2011)
3. Cattani, S., Segala, R.: Decision Algorithms for Probabilistic Bisimulation. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 371–386. Springer, Heidelberg (2002)
4. Chehaibar, G., Garavel, H., Mounier, L., Tawbi, N., Zulian, F.: Specification and Verification of the PowerScale$^{\mathrm{TM}}$ Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In: FORTE, pp. 435–450 (1996)
5. Crouzen, P., Lang, F.: Smart Reduction. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 111–126. Springer, Heidelberg (2011)
6. Deng, Y., Hennessy, M.: On the Semantics of Markov Automata. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 307–318. Springer, Heidelberg (2011)
7. Eisentraut, C., Hermanns, H., Zhang, L.: On Probabilistic Automata in Continuous Time. Reports of SFB/TR 14 AVACS 62, SFB/TR 14 AVACS, long version of LICS 342–351 (2010)
8. Fernandez, J.-C., Mounier, L.: A Tool Set for Deciding Behavioral Equivalences. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 23–42. Springer, Heidelberg (1991)
9. Hermanns, H., Katoen, J.-P.: Automated Compositional Markov Chain Generation for a Plain-Old Telephone System. Science of Computer Programming 36(1), 97–127 (2000)
10. Hermanns, H., Turrini, A.: Deciding Probabilistic Automata weak Bisimulation in Polynomial Time. In: FSTTCS, pp. 435–447 (2012)
11. Kanellakis, P.C., Smolka, S.A.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In: PODC, pp. 228–240 (1983)
12. Kwiatkowska, M.Z., 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)

13. Larsen, K.G., Skou, A.: Bisimulation through Probabilistic Testing (Preliminary Report). In: POPL, pp. 344–352 (1989)
14. Lynch, N.A., Segala, R., Vaandrager, F.W.: Observing Branching Structure through Probabilistic Contexts. SIAM Journal on Computing 37(4), 977–1013 (2007)
15. Milner, R.: Communication and Concurrency. Prentice-Hall International (1989)
16. Paige, R., Tarjan, R.E.: Three Partition Refinement Algorithms. SIAM Journal on Computing 16(6), 973–989 (1987)
17. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, MIT (1995)
18. Segala, R., Lynch, N.A.: Probabilistic Simulations for Probabilistic Processes. Nordic Journal of Computing 2(2), 250–273 (1995)

LTL Model Checking of Interval Markov Chains

Michael Benedikt, Rastislav Lenhardt, and James Worrell

Department of Computer Science, University of Oxford, United Kingdom

Abstract. Interval Markov chains (IMCs) generalize ordinary Markov chains by having interval-valued transition probabilities. They are useful for modeling systems in which some transition probabilities depend on an unknown environment, are only approximately known, or are parameters that can be controlled. We consider the problem of computing values for the unknown probabilities in an IMC that maximize the probability of satisfying an ω-regular specification. We give new upper and lower bounds on the complexity of this problem. We then describe an approach based on an expectation maximization algorithm. We provide some analytical guarantees on the algorithm, and show how it can be combined with translation of logic to automata. We give experiments showing that the resulting system gives a practical approach to model checking IMCs.

1 Introduction

Interval Markov chains (IMCs) generalize ordinary Markov chains by allowing undetermined transition probabilities that are constrained to intervals [14]. IMCs arise naturally in the modelling and verification of probabilistic systems. For example, some transition probabilities may depend on an unknown environment, may only be approximately known, or may be parameters that can be optimized.

Interval Markov chains can be seen as a type of Markov decision process. Valuations of their undetermined transition probabilities can correspondingly be seen as history-free stochastic schedulers. This enforced history-independence makes the theory of IMCs different from that of MDPs. In this paper we consider the problem of computing the optimal (either maximum or minimum) probability that an IMC can satisfy some target specification, where the latter is given as an automaton or as a Linear Temporal Logic (LTL) formula. In previous work on verifying IMCs, Chatterjee *et al.* [7] focus on branching-time properties and Delahaye *et al.* [10] consider refinement. While [7] obtain a 2EXPTIME bound for LTL as a consequence of their results, algorithms and complexity bounds for basic linear-time problems on IMCs have, to the best of our knowledge, not been studied in their own right.

We begin with a study of the complexity of optimizing IMCs with respect to linear-time specifications. We give new upper bounds on the reachability problem and the model checking problem for deterministic automata, unambiguous automata, and LTL. We also show that the 2EXPTIME upper-bound from [7] for LTL can be improved to EXPSPACE in general and to PSPACE when the number of parameters is fixed. We complement this with new lower-bounds, showing that solving the optimization problem for unambiguous automata within the polynomial hierarchy would have significant consequences for the complexity of fundamental problems in symbolic computation

We then turn to practical algorithms for LTL model-checking of IMCs. We use the *expectation-maximization* procedure, which is ubiquitous in machine learning. Indeed, our algorithm can be seen as a variant of the classical Baum-Welch procedure, which finds the optimal probability that an IMC generates a fixed set of sample data. The Baum-Welch procedure progressively re-estimates values of the parameters, giving relatively greater weight to transitions that occur frequently on computations that satisfy a desired property. Analogously with Baum-Welch, we show that our algorithm converges, but not necessarily to the value of the optimal parameters. Our solution to LTL model checking of IMCs couples the expectation-maximization algorithm with a translation of LTL to unambiguous automata. We show that the approach works well in practice, and allows one to take-advantage of the use of unambiguous automata as an intermediate representation.

In summary, our contributions are: (i) Improved upper bounds for model-checking of IMCs with respect to linear-time problems; (ii) New lower bounds, which give new insight into the expressiveness of IMCs; (iii) A novel algorithmic approach to solving the model checking problem in practice; (iv) Experimental results comparing both our LTL translation methods and our end-to-end solution to other techniques. For space reasons, some proofs are omitted.

2 Definitions

Logic and Automata. We specify ω-regular properties using *Linear Temporal Logic* LTL and *Büchi automata.* The formulas of LTL are built from atomic propositions using Boolean connectives and the temporal operators $\bigcirc$ (*next*), $\mathcal{U}$ (*until*) and $\mathcal{R}$ (*release*). Formally, LTL is defined by the following grammar:

$$\varphi ::= p_i \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi\,\mathcal{R}\,\varphi \mid \bigcirc\varphi\,,$$

where $p_0, p_1, \ldots$ are propositional variables. We abbreviate true $\mathcal{U}\ \varphi$ as $\Diamond\varphi$ and write $\Box\varphi$ for $\neg\Diamond\neg\varphi$. We refer the reader to [17] for the semantics of LTL.

A *generalized Büchi automaton* $\mathcal{A}$ is a tuple $(\Sigma, Q, Q_0, \Delta, \mathcal{F})$ with alphabet Σ, set of states Q, set of initial states $Q_0 \subseteq Q$, transition relation $\Delta \subseteq Q \times \Sigma \times Q$, and a collection of accepting sets $\mathcal{F} = \{F_1, \ldots, F_k\}$, where $F_i \subseteq Q$. An infinite run of $\mathcal{A}$ is *accepting* if each set $F \in \mathcal{F}$ is visited infinitely often in the run. We say that $\mathcal{A}$ is *unambiguous* if each word has at most one accepting run.

Interval Markov Chains. A Markov chain is a tuple $\mathcal{M} = (S, \pi_0, M)$, where S is a finite set of states, π_0 is the initial-state distribution on S, and $M : S \times S \to [0, 1]$ is a stochastic transition matrix, i.e., $\sum_{t\in S} M(s, t) = 1$ for all $s \in S$. $\mathcal{M}$ induces a Borel probability measure $Pr_{\mathcal{M}}$ on S^ω in the standard way. An *interval Markov chain* is a tuple $\mathcal{M} = (S, \pi_0, M_l, M_u)$ in which the transition matrix of a Markov chain is replaced with two matrices $M_u, M_l : S \times S \to [0,1]$ with $M_l \leq M_u$. Intuitively M_l and M_u give respective lower and upper bounds on the transition probabilities. An *incomplete Markov chain* is a special case of an interval Markov chain in which for each pair of states s, t, either $M_l(s,t) = M_u(s,t)$ or $M_l(s,t) = 0$ and $M_u(s,t) = 1$, that is, a probability is either precisely given or completely unspecified. An interval Markov chain $\mathcal{M} = (S, \pi_0, M_l, M_u)$ is *refined* by a Markov chain $\mathcal{M}' = (S, \pi_0, M)$ if $M_l(s,t) \leq M(s,t) \leq M_u(s,t)$ for all pairs of states $s, t \in S$. Note that $\mathcal{M}'$ has the same set of states and initial distribution as $\mathcal{M}$.

Given an interval Markov chain $\mathcal{M}$ with set of states S and a labelling function $V : S \to \Sigma$, we want to compute a Markov chain refining $\mathcal{M}$ that optimizes the probability of satisfying a given ω-regular property $L \subseteq \Sigma^\omega$. We call this the *IMC model checking problem*. We will focus in this paper on the case of maximizing the probability of L, but it is easy to modify the techniques to get minimisation. When investigating the complexity of this problem, we will deal with the corresponding decision problem: whether the optimal probability is above a given rational threshold. Let us also note immediately that the problem can be simplified, without loss of generality, by assuming that $\Sigma = S$ and that V is the identity function, i.e., that L is an ω-regular set of trajectories of the IMC. We can do this because ω-regular languages are closed under inverse images of alphabet renamings. We will also use the term *qualitative model checking problem* to refer to the question of whether the probability to satisfy the property L can be made 1.

Product Construction. Next we recall the product construction for Markov chains with unambiguous Büchi automata, which has been noted in several prior works (see, e.g., [8]). An advantage of working with unambiguous automata rather than deterministic automata is that there is a singly exponential translation of LTL to unambiguous automata, whereas the translation of LTL to deterministic automata is doubly exponential.

Let $\mathcal{M} = (S, \pi_0, M)$ be a Markov chain and $\mathcal{A} = (S, Q, Q_0, \Delta, F)$ an unambiguous Büchi automaton whose input alphabet is the set S of states of $\mathcal{M}$. Define the *product graph* $\mathcal{G}_{\mathcal{M}\otimes\mathcal{A}} = (V, E)$ to have set of vertices $V = S \times Q$ and set of edges $E = \{((s,q),(s',q')) : M(s,s') \rangle 0 \text{ and } (q,s',q') \in \Delta\}$.

A strongly connected subset C of $\mathcal{G}_{\mathcal{M}\otimes\mathcal{A}}$ is said to be *accepting* if: (i) for each vertex $(s,q) \in C$, s lies in a bottom strongly connected component of $\mathcal{M}$; (ii) for each vertex $(s,q) \in C$ and edge (s,s') in $\mathcal{M}$ there exists an edge (q,s',q') in $\mathcal{A}$ with $(s',q') \in C$; (iii) for each accepting set $F \in \mathcal{F}$ there exists a vertex $(s,q) \in C$ such that $q \in F$. By extension, a vertex of $\mathcal{G}_{\mathcal{M}\otimes\mathcal{A}}$ is said to be *accepting* if it lies in an accepting set. A vertex is said to be *dead* if it has no path to an accepting vertex. Write V_{yes} for the set of accepting vertices, V_{no} for the set of dead vertices, and $V_?$ for the remaining vertices. Finally, we say that an infinite path in V^ω is *accepting* if it has a tail consisting exclusively of accepting vertices.

We can define probabilistic transitions on the product graph, with a transition from (s, q) to (s', q') being given the value:

$$M'((s,q),(s',q')) = \begin{cases} M(s,s') & ((s,q),(s',q')) \in E \text{ and } (s',q') \in V_{yes} \cup V_? \\ 0 & \text{otherwise} \end{cases}$$

We also define an initial probability vector $\pi'_0 \in \mathbb{R}^V$ by

$$\pi'(s,q) = \begin{cases} \pi_0(s) & (p,s,q) \in \varDelta \text{ for some } p \in Q_0 \\ 0 & \text{otherwise} \end{cases}$$

Since $\mathcal{A}$ is non-deterministic, M' need not correspond to a stochastic matrix and π' need not be a probability distribution. Nevertheless M' and π' induce a Borel sub-probability measure $Pr_{\mathcal{M}\otimes\mathcal{A}}$ on V^ω by defining

$$Pr_{\mathcal{M}\otimes\mathcal{A}}(C(v_1 \ldots v_n)) = \pi'(v_1) \cdot M'(v_1,v_2) \cdot M'(v_2,v_3) \cdots M'(v_{n-1},v_n)$$

where $C(v_1 \ldots v_n)$ is the *cylinder* set of words in V^ω with prefix $v_1 \ldots v_n$.

Write $\Diamond V_{yes} \subseteq V^\omega$ for the set of infinite paths that contain an accepting vertex. The following result allows us to reduce the model checking problem for $\mathcal{M}$ and $\mathcal{A}$ to calculating the probability of reaching an accepting vertex in the product graph, which can be done using linear algebra. We can then verify that: $Pr_{\mathcal{M}\otimes\mathcal{A}}(\Diamond V_{yes}) = Pr_{\mathcal{M}}(L(\mathcal{A}))$.

3 Complexity of Verification Problems

Reachability for IMCs is more involved than for MDPs since the interval constraints preclude restricting to deterministic schedulers. As with MDPs we can reduce reachability to linear programming. The resulting linear program is exponential in the size of the IMC, but it has a polynomial-time separation oracle and can therefore be solved in polynomial time using the ellipsoid method.

Proposition 1. *Reachability in interval Markov chains is P-complete.*

Proof. The lower bound is via reduction from the monotone circuit value problem, with the argument identical to P-hardness of computing optimal strategies for reachability in MDPs [16]. The lower bound holds even for incomplete Markov chains.

For the upper bound, we reason as follows. Let $\mathcal{M} = (S, \pi_0, M_l, M_u)$ be an IMC. We can identify the set of refinements of $\mathcal{M}$ with the convex set $[\![\mathcal{M}]\!] \subseteq \mathbb{R}^{S\times S}$ of stochastic transition matrices M such that $M_l(s,t) \leq M(s,t) \leq M_u(s,t)$ for all $s,t \in S$. Observe that M is a vertex of $[\![\mathcal{M}]\!]$ if and only if for each state s at most one of the outgoing transition probabilities $M(s,t)$ is strictly between its lower bound $M_l(s,t)$ and its upper bound $M_u(s,t)$.

Consider the following linear program with variables $\boldsymbol{x} = \{x_s : s \in S\}$.

$$\begin{array}{ll} \text{minimise } \sum_{s\in S} x_s & \\ \text{subject to} & \\ \boldsymbol{x} \geq M\boldsymbol{x} & \text{for each vertex } M \text{ of } [[\mathcal{M}]] \\ \boldsymbol{x} \geq \chi_F & \end{array}$$

where χ_F is the characteristic vector of the set F.

By convexity we observe that $\boldsymbol{x} \geq \chi_F$ is feasible for the above program if and only if $\boldsymbol{x} \geq M\boldsymbol{x}$ for all transition matrices $M \in [[\mathcal{M}]]$. Thus $\boldsymbol{x}$ is feasible if and only if x_s is an upper bound for the probability to reach F from state s for all refinements of $\mathcal{M}$. We conclude that the optimal solution of the linear program gives the maximum probability to reach F over all refinements.

Although the number of constraints in this linear program is exponential in the number of states of $\mathcal{M}$, we do not need their explicit representation. We can use the Ellipsoid algorithm [13] to find the optimal values of x_s in polynomial time. The Ellipsoid algorithm needs an oracle to determine whether given values of x_s are feasible, and, if not, output a separating hyperplane, i.e., the inequality that does not hold. In fact, given a family of values x_s it suffices to consider a single "dominating" constraint in the above program, namely the transition matrix M that simultaneously maximises each entry of $M\boldsymbol{x}$. This matrix is easy to compute: Let $s_1, s_2, \ldots$ be an enumeration of S such that $x_{s_1} \geq x_{s_2} \geq \ldots$. Now for each state $s \in S$, choose $M(s, s_1)$ as high as possible (compatible with all other edges achieving their lower bounds); if $M(s, s_1)$ is at its upper bound, we set $M(s, s_2)$ as high as possible, etc. $\square$

We now turn to verification of properties given as unambiguous Büchi automata. Note that we can not apply a product construction to reduce to the reachability problem for IMCs; the natural product would have the same variable repeated many times in the product chain, introducing correlation. We introduce a practical technique for addressing this problem in the following section. Still, we can get a polynomial space upper bound by reduction to the decision problem for the existential theory of the reals [6].

Theorem 1. *The model-checking problem for unambiguous Büchi automata on IMCs is in PSPACE.*

A matching PSPACE lower bound in Theorem 1 would imply PSPACE-hardness of the decision problem for the existential theory of the reals, which is open. However we can precisely characterise the complexity of the model checking problem for IMCs against unambiguous Büchi automata in terms of the Blum-Shub-Smale (BSS) model of computation over the real field with order $(\mathbb{R}, \leq)$ [4]. In this model each tape cell of a Turing machine can hold a single real number and a decision problem is a language $L \subseteq \mathbb{R}^*$. Arithmetic operations and sign tests have unit cost regardless of the operands, otherwise the classes of polynomial-time problems, denoted $\mathrm{P}_{\mathbb{R}}$, and non-deterministic polynomial-time problems, denoted $\mathrm{NP}_{\mathbb{R}}$, are defined analogously with the classical case. Note that in the definition of $\mathrm{NP}_{\mathbb{R}}$ the "certificate" is a polynomial-length string of real numbers. We now show:

Theorem 2. *The model checking problem for interval Markov chains with respect to unambiguous automata is $NP_{\mathbb{R}}$-complete.*

Proof. The upper bound is easy given that $\mathrm{NP}_{\mathbb{R}}$ allows the guessing of real numbers, which is precisely what is needed in the model checking problem. The lower bound is via reduction from the problem $(0,1)$-Pos, whose input consists of a real polynomial f and threshold $\theta \in (0,1)$, the output being yes iff there exist values for the variables of f lying in the open interval $(0,1)$ such that $f \geq \theta$. We assume that f is presented as a sum of products of constants $\alpha \in (0,1)$ and *literals* $x, 1-x$, where x is a variable. $(0,1)$-Pos can be shown hard for $\mathrm{NP}_{\mathbb{R}}$ via reduction from the known hard problem of determining whether a polynomial of degree at most 4 has a real root. We first give a brief overview of the reduction from $(0,1)$-Pos to IMC model checking.

Given an instance f, θ of $(0,1)$-Pos, we build an IMC $\mathcal{M}$ with nodes corresponding to constants and variables of f, along with nodes that designate whether a variable x is to be complemented (transformed into $1-x$). We also build a regular expression E so that the probability E can take over $\mathcal{M}$ as a function of the variables of $\mathcal{M}$ corresponds exactly to f. Then the problem $f \geq \theta$ translates to the problem of model checking E on $\mathcal{M}$.

Let $f \geq \theta$ be an instance of $(0,1)$-Pos, where f mentions real constants $\alpha_1, \ldots, \alpha_m$ and variables $x_1, \ldots, x_n$. We derive an incomplete Markov chain $\mathcal{M} = (S, \pi_0, M)$ from f as follows. The set of states is $S = \{c_1, \ldots, c_{n+m}\} \cup \{h, t\}$, with initial distribution π_0 the uniform distribution on $\{c_1, \ldots, c_{m+n}\}$. We think of each state c_i as a biased coin that represents either a constant or a variable. States $c_1, \ldots, c_m$ represent the constants, and accordingly we define fixed transition probabilities $M(c_i, h) = \alpha_i$ and $M(c_i, t) = 1 - \alpha_i$ for $1 \leq i \leq m$. States $c_{m+1}, \ldots, c_{m+n}$ represent the variables, and we leave the transition probabilities $M(c_i, h)$ and $M(c_i, t)$ undefined. We define $M(h, c_i) = M(t, c_i) = 1/(n+m)$ for all $1 \leq i \leq n+m$. All other transition probabilities are zero.

We define a mapping φ from the constants and literals occurring in f to edges of $\mathcal{M}$ by $\varphi(\alpha_i) = c_i h$, $\varphi(x_i) = c_{i+m} h$, and $\varphi(1 - x_i) = c_{i+m} t$. Write $f = \sum_{i=1}^{k} \prod_{j=1}^{l} f_{i,j}$, where each $f_{i,j}$ is a constant or literal. (We can assume that each product has the same number of terms l by suitable padding.) Then we define a regular expression $E = \sum_{i=1}^{k} \prod_{j=1}^{l} \varphi(f_{i,j})$ over alphabet S, the set of states of $\mathcal{M}$. We can further identify a Markov chain $\mathcal{M}'$ refining $\mathcal{M}$ with a valuation of the variables occurring in f, where variable x_i gets the transition probability p_i to go from c_i to h. Under this identification it is easy to see that

$$Pr_{\mathcal{M}'}(ES^{\omega}) = \frac{f(p_1, \ldots, p_n)}{(n+m)^{l+1}} .$$

This equation straightforwardly allows us to reduce a positivity query on f to a model checking query on $\mathcal{M}$. Note that the requirement in $(0,1)$-Pos that variables only take values in $(0,1)$ can be enforced by modifying the specification language to contain only strings with infinitely many occurrences of $c_i h$ for every i. Finally, it is straightforward to represent the specification language as a deterministic automaton of polynomial size. This completes the proof of Theorem 2. $\square$

The classes $\mathrm{P}_{\mathbb{R}}$ and $\mathrm{NP}_{\mathbb{R}}$ can be compared to classical Boolean complexity classes by considering their *Boolean parts*. The Boolean part of a complexity class $\mathcal{C}$ in the BSS model is defined to be $\mathrm{BP}(\mathcal{C}) = \{L \cap \{0,1\}^* : L \in \mathcal{C}\}$. It is well known that NP is contained in $\mathrm{BP}(\mathrm{NP}_{\mathbb{R}})$ [4] and that PosSLP is contained in $\mathrm{BP}(\mathrm{P}_{\mathbb{R}})$ [2]. (Recall that PosSLP is the problem of determining whether an arithmetic circuit with integer inputs evaluates to a positive number [2]) It follows from Theorem 2 that the model checking problem for IMCs against unambiguous Büchi automata is both NP-hard and PosSLP-hard. The NP lower bound is already known in the form of NP-hardness of the maximum-likelihood problem for hidden Markov chains [1].

Finally, we turn to LTL model-checking. Formerly, the only known upper bound for LTL model-checking of IMCs was 2EXPTIME [7], the same as for general MDPs. Below we note that a better bound of EXPSPACE can be obtained. More interestingly, if the number of parameters is fixed, the complexity reduces to PSPACE.

Theorem 3. *The LTL model checking problem for IMCs is in EXPSPACE, and is PSPACE-hard. For fixed parameters, the problem is PSPACE-complete. The qualitative model-checking problem is PSPACE-complete.*

Proof. We consider interval Markov chains with a fixed number k of undetermined transition probabilities. We represent these probabilities by variables $x_1, \ldots, x_k$ and work with the field of rational functions $\mathbb{F} = \mathbb{Q}(x_1, \ldots, x_k)$.

Let $\mathcal{M}$ be an interval Markov chain and φ an LTL formula with respective sizes $||\mathcal{M}||$ and $||\varphi||$. Using polynomial space in $||\mathcal{M}||$ and $||\varphi||$ one can translate φ into an equivalent unambiguous Büchi automaton $\mathcal{A}$, build the product graph $\mathcal{G}_{\mathcal{M}\otimes\mathcal{A}}$, and derive a corresponding system of linear equations with coefficients in $\mathbb{F}$ whose solution is an element of $\mathbb{F}$ that represents $P_{\mathcal{M}}(L(\mathcal{A}))$ as a function of $x_1, \ldots, x_k$. This system of equations has size exponential in $||\mathcal{M}||$ and $||\varphi||$.

Now systems of linear equations with coefficients in $\mathbb{F}$ can be solved in polylogarithmic space [5]. Thus, using polynomial space in $||\mathcal{M}||$ and $||\varphi||$ overall, we can compute a rational function $f(x_1, \ldots, x_k) \in \mathbb{F}$ that represents the probability $P_{\mathcal{M}}(L(\mathcal{A}))$. Again, the expression representing f has size exponential in $||\mathcal{M}||$ and $||\varphi||$. Finally we use the polylogarithmic-space procedure of Ben-Or, Kozen and Reif [3] for deciding satisfiability of quantifier-free formulas in the first-order theory of real-closed fields over the fixed set of variables $x_1, \ldots, x_k$. With this procedure we can test the existence of transition probabilities $x_1, \ldots, x_k$ such that $P_{\mathcal{M}}(L(\mathcal{A}))$ is greater than a given threshold using overall space that is polynomial in $||\mathcal{M}||$ and $||\varphi||$. □

4 Expectation Maximization Algorithm

In this section we describe an expectation maximization algorithm that, given an initial refinement $\mathcal{M}_0$ of an IMC $\mathcal{M}$, produces a sequence of refinements having successively higher probabilities of satisfying a given ω-regular property, presented as an unambiguous Büchi automaton $\mathcal{A}$. We assume initially that $\mathcal{M}$ is an incomplete Markov chain and discuss the more general case of interval Markov chains later. We also defer until later a discussion of how the initial refinement is chosen.

Overview. Figure 1 gives an outline of the algorithm. The input includes a parameter n governing the number of iterations of the update procedure. The intuitive idea of the update procedure is to assign relatively greater weight to transitions that are most likely to be taken in computations of the current refinement $\mathcal{M}_i$ that are accepted by $\mathcal{A}$.

```
Algorithm EM
Input: Incomplete Markov chain $\mathcal{M} = (S, \pi_0, M_l, M_u)$, Initial refinement $\mathcal{M}_0$
Unambiguous Büchi automaton $\mathcal{A} = (S, Q, Q_0, \Delta, \mathcal{F})$, Iteration parameter $n$
Begin
For $i = 0$ to $n - 1$ do
    $\mathcal{M}_{i+1} := \mathit{update}(\mathcal{M}_i)$
End
```

Fig. 1. EM Algorithm

The Update Procedure. We now explain in more detail the operation of the update procedure. Assume that we are given a refinement $\mathcal{M}_i$ of $\mathcal{M}$ with associated product graph $\mathcal{G}_{\mathcal{M}_i \otimes \mathcal{A}} = (V, E)$. Write M_i for the transition matrix of $\mathcal{M}_i$ and write π'_0 and M'_i for the lifting of the initial state distribution and transition matrix of $\mathcal{M}_i$ to the product $\mathcal{G}_{\mathcal{M}_i \otimes \mathcal{A}}$, as defined in Section 2. Given an infinite path $v_1 v_2 v_3 \ldots \in V^\omega$, say that $v_1 \ldots v_n$ is a *minimum accepting prefix* if $v_i \in V_?$ for $1 \leq i \leq n-1$ and $v_n \in V_{yes}$, i.e., v_n is the first accepting vertex on the path.

Write $U \subseteq S \times S$ for the set of pairs (s, t) of states of the incomplete Markov chain $\mathcal{M}$ whose transition probability is undetermined. For each pair of Markov-chain states $(s, t) \in U$ we define a random variable $Z_{s,t} : V^\omega \to \mathbb{N}$ that takes value 0 on any non-accepting path in $\mathcal{G}_{\mathcal{M}_i \otimes \mathcal{A}}$ and otherwise equals the number of occurrences of edge (s, t) in the projection onto $\mathcal{M}$ of the minimum accepting prefix of the path. The update procedure is based on computing $\mathbf{E}[Z_{s,t}]$.

For each pair $(s, t) \in U$ we compute $\mathbf{E}[Z_{s,t}]$ using a variant of the classical *forward-backward* algorithm for hidden Markov models. Given a vertex $(s, q) \in V_?$, define $\alpha(s, q)$ to be the expected value of the random variable that maps each non-accepting path of $\mathcal{G}_{\mathcal{M}_i \otimes \mathcal{A}}$ to 0 and maps each accepting path to the number of occurrences of (s, q) in a minimum accepting prefix of the path. We can compute $\alpha(s, q)$ as the solution to the following system of linear equations:

$$\alpha(s,q) = \begin{cases} \pi'_0(s,q) + \displaystyle\sum_{(t,p) \in V} (\alpha(t,p) + 1)\, M'_i((t,p),(s,q)) & (s,q) \in V_? \\ 0 & (s,q) \notin V_? \end{cases}$$

We further define $\beta(s, q)$ to be the probability to reach an accepting state in $\mathcal{G}_{\mathcal{M}_i \otimes \mathcal{A}}$ starting at state (s, q). We can compute $\beta(s, q)$ as the solution to the following system of linear equations:

$$\beta(s,q) = \begin{cases} \sum_{(t,p)} M'_i((s,q),(t,p))\, \beta(t,p) & (s,q) \in V_? \\ 1 & (s,q) \in V_{yes} \\ 0 & (s,q) \in V_{no} \end{cases}$$

Finally for each pair $(s,t) \in U$ we define

$$\mathbf{E}[Z_{s,t}] = \sum_{p \in Q} \sum_{q \in Q} \alpha(s,p)\, M_i'((s,p),(t,q))\, \beta(t,q)\,.$$

Furthermore, for a given state $s \in S$, define $\mu_s = \sum\{M_i(s,t) : (s,t) \notin U\}$ to be the total mass of all fixed transition probabilities at state s. The update procedure assigns to $(s,t) \in U$ the transition probability

$$(1-\mu_s) \cdot \frac{\mathbf{E}[Z_{s,t}]}{\sum_{u:(s,u)\in U} \mathbf{E}[Z_{s,u}]} \tag{1}$$

if $(s,q) \notin V_{no}$ for some $q \in Q$. If $(s,q) \in V_{no}$ for all $q \in Q$ then the weight of each edge $(s,t) \in U$ is left unchanged. Thus the new transition probability is determined by the proportion of times that an accepting trajectory of the Markov chain takes the edge (s,t) among all visits to state s before reaching an accepting state of the product.

The following result is proven in the full version.

Theorem 4. *The sequence $Pr_{\mathcal{M}_i}(L(\mathcal{A}))$ is monotonically increasing.*

The choice of initial refinement $\mathcal{M}_0$ can have a significant effect on the behaviour of the EM algorithm. In particular, the choice of which transition values in $\mathcal{M}_0$ are positive governs the initial classification of vertices of the product graph as either accepting or dead. Note that successive iterations of the update procedure do not alter the set of dead vertices. Of course, the connectivity properties of $\mathcal{M}_0$ may be independent of the undefined transition values in an incomplete Markov chain. Furthermore, the choice of positive transitions in $\mathcal{M}_0$ may be suggested by the problem instance. For example, in a repair problem we start with a Markov chain $\mathcal{M}$ and see if it can be made to satisfy a given property by optimizing its transition values within certain intervals. In this case it is natural to take $\mathcal{M}$ itself as the initial refinement. In general, to gain confidence that we have reached the global optimum, we can employ standard heuristics, such as random restart.

We have described the algorithm for incomplete Markov chains, and in the full version we prove that it progressively improves the expectation and converges to a local maximum. For IMCs an update may violate the restricted ranges for intervals. In this case we consider the output of the update algorithm for every refinement of the IMC formed by fixing some parameters to be at the boundary. At least one of these must be feasible (e.g. those where all parameters are fixed), and we choose the refinement that gives the optimal probability.

LTL to Unambiguous Büchi Automata. To apply the expectation maximization algorithm to LTL formulas, we use a translation from LTL to unambiguous generalized Büchi automata. Our approach is a modification of the build-by-need construction of Gerth *et al.* [12] in which we adjust the translation rules from Section 3.2 of [12] to remove potential ambiguity. For example, one step of the [12] procedure involves splitting an automaton state labelled with the subformula $\varphi \vee \psi$ into two copies, one labelled φ and the other labelled ψ. In our approach such a state is instead split into a copy labelled φ and a copy labelled $\neg\varphi \wedge \psi$. The mutual exclusivity of the logical formulas leads to the production of an unambiguous automaton. The operators $\mathcal{U}$ and $\mathcal{R}$ are treated in similar fashion (see below).

Formula	[12] splits to		Tulip splits to	
$\varphi \vee \psi$	φ	ψ	φ	$\neg\varphi \wedge \psi$
$\varphi\, \mathcal{U}\, \psi$	ψ	$\varphi \wedge \bigcirc(\varphi\, \mathcal{U}\, \psi)$	ψ	$\neg\psi \wedge \varphi \wedge \bigcirc(\varphi\, \mathcal{U}\, \psi)$
$\varphi\, \mathcal{R}\, \psi$	$\varphi \wedge \psi$	$\psi \wedge \bigcirc(\varphi\, \mathcal{R}\, \psi)$	$\varphi \wedge \psi$	$\neg\varphi \wedge \psi \wedge \bigcirc(\varphi\, \mathcal{R}\, \psi)$

Example 1. Consider the incomplete Markov chain $\mathcal{M}$ with undefined transition probabilities, represented by variables x and y, shown in Figure 2. We optimise $\mathcal{M}$ with respect to the LTL formula $\varphi \stackrel{\text{def}}{=} \Diamond a \wedge \Diamond b$. The automaton $\mathcal{A}$ representing this formula and the product graph $\mathcal{G}_{\mathcal{M}\otimes\mathcal{A}}$ (with accepting vertices and transition probabilities) are also shown in Figure 2.

Considering $\mathcal{G}_{\mathcal{M}\otimes\mathcal{A}}$, the expected number of times for a run starting in vertex v_1 to visit vertices v_4 and v_5 is given by

$$\alpha(v_4) = \sum_{i=1}^{\infty} \left(\frac{x}{10}\right)^i = \frac{x}{10-x} \quad \text{and} \quad \alpha(v_5) = \sum_{i=1}^{\infty} \left(\frac{4y}{5}\right)^i = \frac{4y}{5-4y}$$

Furthermore, let $\beta(v)$ be the probability to reach an accepting vertex from v. Then we have $\beta(v_2) = \frac{y}{10-x}$ and $\beta(v_3) = \frac{4x}{5-4y}$.

From (4) the expected number of times to take the x-labelled edge in $\mathcal{M}$ along a run that satisfies $\Diamond a \wedge \Diamond b$ is

$$\begin{aligned} f(x) &\stackrel{\text{def}}{=} \alpha(v_1)\cdot x \cdot \beta(v_2) + \alpha(v_4)\cdot x\cdot \beta(v_2) + \alpha(v_5)\cdot x\cdot\beta(v_6) \\ &= \frac{xy}{10-x} + \frac{x^2y}{(10-x)^2} + \frac{4yx}{5-4y}\,. \end{aligned}$$

Likewise, the expected number of times to take the y-labelled edge in $\mathcal{M}$ along a run that satisfies $\Diamond a \wedge \Diamond b$ is

$$\begin{aligned} g(x) &\stackrel{\text{def}}{=} \alpha(v_1)\cdot y \cdot \beta(v_3) + \alpha(v_5)\cdot y\cdot \beta(v_3) + \alpha(v_4)\cdot y\cdot\beta(v_6) \\ &= \frac{4xy}{5-4y} + \frac{16xy^2}{(5-4y)^2} + \frac{xy}{10-x}\,. \end{aligned}$$

Now the sequence of transition values (x_n) defined by $x_{n+1} = \frac{f(x_n)}{f(x_n)+g(x_n)}$ converges to a limit ($\simeq 0.32$) that maximizes the probability of satisfying $\Diamond a \wedge \Diamond b$.

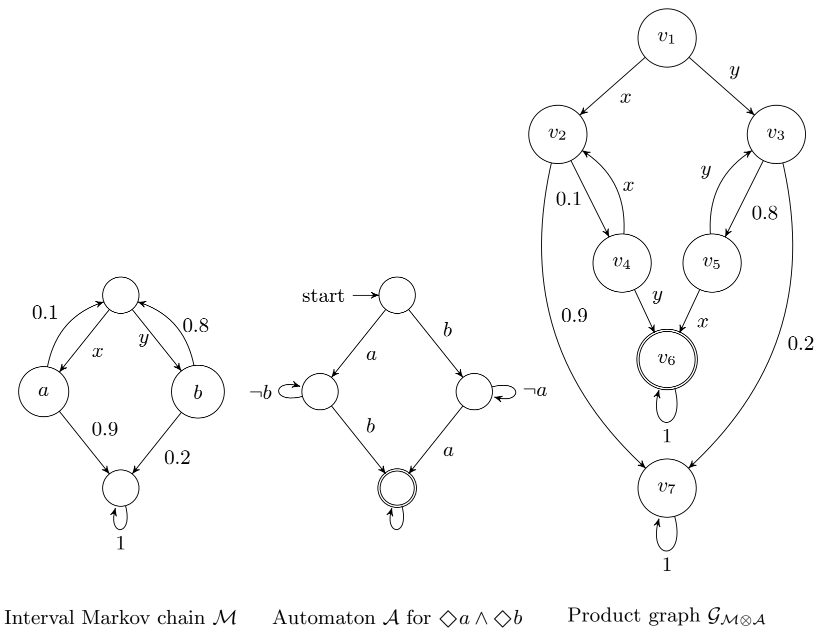

0.1
0.8
x
y
a
b
0.9
0.2
1
Interval Markov chain $\mathcal{M}$
start
a
b
$\neg b$
$\neg a$
b
a
Automaton $\mathcal{A}$ for $\Diamond a \wedge \Diamond b$
v_1
x
y
v_2
v_3
0.1
x
y
0.8
v_4
v_5
y
x
0.9
v_6
0.2
1
v_7
1
Product graph $\mathcal{G}_{\mathcal{M}\otimes\mathcal{A}}$

Fig. 2. Example

5 Implementation and Experiments

Our tool Tulip can be accessed at http://tulip.lenhardt.co.uk along with several examples. The tool inputs a labelled interval Markov chain along with properties specified either by LTL formulas or directly by unambiguous Büchi automata. It performs a specified number of iterations of the EM algorithm and outputs an approximation to the maximum probability with which the IMC satisfies the property, together with the values within the intervals for which the maximum is achieved.

LTL-to-Automaton Translation. We begin by comparing the performance of our translation component with other methods of generating automata from LTL. Our translation begins by pre-processing the formula using the simplifier of LTL2dstar [15], allowing us, for example, to notice that LTL formula $\neg(\Diamond\Diamond p_1 \leftrightarrow \Diamond p_1)$ is equivalent to *false*. The table below compares the unambiguous automata we construct with the experimental results of constructing non-deterministic automata reported in [12]. We took formulas from [12] covering a range of useful properties, such as fairness. We compare with [12] because the non-deterministic automata generated by [12] are already extremely small, as shown below. Our experiments suggest that the extra cost of producing unambiguous automata is usually very small, and so using Tulip we get nearly optimal unambiguous automata. This is encouraging given that unambiguous automata can be used directly for probabilistic model checking without determinization.

Moreover, assuming reasonable computational resources (1 GB of RAM and few seconds of CPU time), we were able to use Tulip to construct automata with up to $10,000$ nodes.

	[12]		Tulip	
Formula	Nodes	Edges	Nodes	Edges
$p_1 \,\mathcal{U}\, p_2$	3	4	3	5
$p_1 \,\mathcal{U}\, (p_2 \,\mathcal{U}\, p_3)$	4	6	4	9
$\neg(p_1 \,\mathcal{U}\, (p_2 \,\mathcal{U}\, p_3))$	7	15	5	12
$\Box\Diamond p_1 \rightarrow \Box\Diamond p_2$	9	15	8	18
$\Diamond p_1 \,\mathcal{U}\, \Box p_2$	8	15	6	16
$\Box(p_1 \,\mathcal{U}\, p_2)$	5	6	4	10
$\neg(\Diamond\Diamond p_1 \leftrightarrow \Diamond p_1)$	22	41	1	1

In general, non-deterministic automata can be exponentially more succinct than unambiguous automata. There are also cases, such as e.g. LTL formula $\Diamond(a \wedge \bigcirc^k a)$, when Tulip translates the formula into an automaton with number of states exponential in k. However, as our comparison with PRISM illustrates below, even in this case Tulip can still produce much smaller automata by avoiding the need to determinize.

Optimizations on Product Chains. Here we describe some of the optimizations that we use to reduce the state space of automata and the cross-product of automata and IMCs. We apply probabilistic bisimulation to the cross product, extending the usual notion to handle parameters. In a step of an iterative refinement algorithm for (standard) probabilistic bisimulation, one has to match the total mass of transitioning from a state u to some equivalence class E with the mass passing from a state v to E. In the case of our cross product machine, our transitions are labeled with parameters from the IMC, and our notion of matching is as a formal sum. We have found that the time spent performing bisimulation was more than compensated for by allowing faster iterations and reduced memory consumption of the EM algorithm,

Another opportunity to reduce the state space is to collapse vertices. First note that when we form a cross product between an IMC and an unambiguous automaton we can determine nodes that are "almost surely accepting" (i.e. starting at this vertex we will accept with probability one), just by checking the underlying structure of the graph. More generally, vertices can be grouped together if almost every accepting path that goes through one must go through another. For example, vertices in a bottom SCC can be collapsed into a single vertex, and linear subgraphs that do not contain accepting vertices can be collapsed.

Benchmarks. To test the effect of our automata translation techniques on the performance of LTL model checking, we consider a simple *Probabilistic Broadcast Protocol (PBP)* [11] by which nodes in a network propagate information. In this protocol when a node receives a message, it broadcasts the message to its neighbours with a certain probability and otherwise ignores the message.

In either case the node then goes to sleep. We model a synchronous variant with message collision: all sending and receiving is in rounds, and if a node is sent a message simultaneously from two neighbours it only receives noise. Tulip imports an existing Markov-chain model of the protocol from PRISM. There are no interval transitions in this model.

We model check the LTL property $\Diamond(a \wedge \bigcirc^k a)$ for various values of the parameter k, where a denotes the sending of a message to a given node in the network. The table below gives the outcome, showing how Tulip outperforms PRISM on this example. We attribute the latter to Tulip's use of unambiguous automata, whereas PRISM relies on a complex determinization construction. The Markov chain in this example is relatively small, so PRISM's symbolic model checking capability is not exploited.

k	2	3	4	5	6	7	8	9	10
Tulip	0.017	0.026	0.065	0.072	0.140	0.292	0.471	0.859	1.412
PRISM	0.015	0.023	0.040	0.111	0.369	0.864	1.820	6.465	30.101

Now we turn to the case of Interval Markov Chains, considering all stages of our algorithm. We evaluate the performance of Tulip using a single core of 1.7 Ghz Intel Core i5 CPU. The first column contains results for the interval Markov chain from Example 1. The second column contains results for model checking the *Bounded Retransmission Protocol (BRP)* [9]. The BRP splits a given file into N chunks and tries to send each of them at most *MAX* times, using two lossy channels for transmissions and acknowledgements. In contrast to prior modeling of this protocol (e.g. in PRISM), we do not model message losses by a fixed probability but by intervals representing a range estimate on their reliability. We set $N = 32$, $MAX = 3$ and model check the property that *the sender does not report a successful transmission.*

Interval Markov chain	Example 1		BRP	
LTL property	$\Diamond(a \wedge \bigcirc^8 b)$		$\Diamond a$	
	Nodes	Time(s)	Nodes	Time(s)
Initial automaton	1026	0.142	5	0.000
Automaton after bisimulation	513	0.035	3	0.002
Naive cross product	2052	0.004	5301	0.142
Product with reachable states only	122		1767	
Product after collapse	74	0.008	610	23.944
Product after bisimulation	72	0.009	544	2.139
One iteration of EM algorithm		0.003		0.934

Each iteration of our algorithm runs in cubic time, so the above techniques reducing the size of the product chain are worthwhile. For example, in our benchmarks it could be seen that an iteration on an example with over 500 nodes took less than a second. In the examples above and below, at most tens of iterations are sufficient to attain a precision up to five decimal places. For example, our algorithm stabilized to this level of accuracy in four iterations for the model from Figure 2 (a solution found to be the correct global maximum by hand analysis); we needed only one iteration for the BRP model.

Below we show the impact of our optimizations on additional examples which are described on our website. They cover a range of scenarios, including finding mixed strategies in some economic games and evaluating properties specifying competing goals.

Examples: *Rendezvous in the Park* (R), *Competing Goals* (G), *Modifying Dice* (D), *Predicting Football* (F), *Probabilistic Broadcast Protocol* (P).

	R	G	D	F	P
Size of interval Markov chain	5	4	7	22	79
Initial automaton size	6	10	10	82	162
Automaton after bisimulation	6	4	9	29	129
Naive cross product	30	20	63	638	10191
Product with reachable states only	20	11	21	171	599
Product after collapse	7	9	12	41	83
Product after bisimulation	6	8	6	15	18
Iterations for 5-decimal-digit precision	14	6	1	12	1
Start to end running time (in seconds)	0.013	0.007	0.010	0.024	0.140

The results support our observations above concerning the size of automata generated, the speed of a particular iteration, and the number of iterations required.

6 Conclusions

In this work we show that the IMC model has advantages in complexity of evaluation over general MDPs. This is reflected in our worst-case bounds, and also at the pragmatic level. We are able to avoid translation to deterministic automata, which is essential to MDP solving for LTL specifications, making do instead with unambiguous automata. We are also able to make use of methods for parameter training from other areas. In this paper we have focused on EM, but in future work we will look at adaptations of other training methods, such as gradient descent.

For specifications given by automata, our $\mathrm{NP}_{\mathbb{R}}$-completeness result shows that the complexity of IMC model-checking lies in PSPACE. Note that a PSPACE-hardness result would imply that satisfiability for the existential theory of the reals is PSPACE-hard, while the complexity of this theory has been open for quite some time. For LTL specifications, our results only isolate the complexity between PSPACE and EXPSPACE. We will look for tighter bounds in future work.

References

1. Abe, N., Warmuth, M.K.: On the computational complexity of approximating distributions by probabilistic automata. Machine Learning 9, 205–260 (1992)
2. Allender, E., Bürgisser, P., Kjeldgaard-Pedersen, J., Miltersen, P.B.: On the complexity of numerical analysis. SIAM J. Comput. 38(5), 1987–2006 (2009)

3. Ben-Or, M., Kozen, D., Reif, J.H.: The complexity of elementary algebra and geometry. JCSS 32(2), 251–264 (1986)
4. Blum, L., Cucker, F., Shub, M., Smale, S.: Complexity and real computation. Springer (1997)
5. Borodin, A., Cook, S.A., Pippenger, N.: Parallel computation for well-endowed rings and space-bounded probabilistic machines. Information and Control 58(1-3), 113–136 (1983)
6. Canny, J.F.: Some algebraic and geometric computations in pspace. In: STOC (1988)
7. Chatterjee, K., Sen, K., Henzinger, T.A.: Model-Checking ω-Regular Properties of Interval Markov Chains. In: Amadio, R. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 302–317. Springer, Heidelberg (2008)
8. Couvreur, J.-M., Saheb, N., Sutre, G.: An Optimal Automata Approach to LTL Model Checking of Probabilistic Systems. In: Vardi, M.Y., Voronkov, A. (eds.) LPAR 2003. LNCS, vol. 2850, pp. 361–375. Springer, Heidelberg (2003)
9. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability Analysis of Probabilistic Systems by Successive Refinements. In: de Luca, L., Gilmore, S. (eds.) PAPM-PROBMIV 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001)
10. Delahaye, B., Larsen, K.G., Legay, A., Pedersen, M.L., Wąsowski, A.: Decision Problems for Interval Markov Chains. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 274–285. Springer, Heidelberg (2011)
11. Fehnker, A., Gao, P.: Formal Verification and Simulation for Performance Analysis for Probabilistic Broadcast Protocols. In: Kunz, T., Ravi, S.S. (eds.) ADHOC-NOW 2006. LNCS, vol. 4104, pp. 128–141. Springer, Heidelberg (2006)
12. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification Testing and Verification (1995)
13. Grötschel, M., Lovász, L., Schrijver, A.: Geometric Algorithms and Combinatorial Optimization, vol. 2. Springer (1993)
14. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS (1991)
15. Klein, J., Baier, C.: Experiments with deterministic ω-automata for formulas of linear temporal logic. Theor. Comput. Sci. 363(2), 182–195 (2006)
16. Papadimitriou, C., Tsitsiklis, J.N.: The complexity of markov decision processes. Math. Oper. Res. 12(3), 441–450 (1987)
17. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115, 1–37 (1994)

Ramsey vs. Lexicographic Termination Proving

Byron Cook[1], Abigail See[2], and Florian Zuleger[3,⋆]

[1] Microsoft Research & University College London
[2] University of Cambridge
[3] TU Wien

Abstract. Termination proving has traditionally been based on the search for (possibly lexicographic) ranking functions. In recent years, however, the discovery of termination proof techniques based on Ramsey's theorem have led to new automation strategies, *e.g.* size-change, or iterative reductions from termination to safety. In this paper we revisit the decision to use Ramsey-based termination arguments in the iterative approach. We describe a new iterative termination proving procedure that instead searches for lexicographic termination arguments. Using experimental evidence we show that this new method leads to dramatic speedups.

1 Introduction

The traditional method of proving program termination (*e.g.* from Turing [25]) is to find a single monolithic ranking function that demonstrates progress towards a bound during each transition of the system. Often, in this setting, we must use lexicographic arguments (*i.e.* ranking functions with range more complex than the natural numbers), as simple linear ranking functions are not powerful enough even in some trivial cases. Recent tools (*e.g.* [3], [8], [14], [15], etc) have moved away from single ranking functions and towards termination arguments based on Ramsey's theorem (*e.g.* [7], [9], [11], [22], etc). The advantage of these new approaches is that we do not need to find lexicographic termination arguments, which are perceived to be difficult to find for large programs. Instead, in these new frameworks, we typically need only to find a set of simple linear ranking functions. The important distinction here is that lexicographic ordering does not matter, thus making the *finding* of termination arguments much easier.

The difficulty with these new termination methods is establishing validity of the termination argument in hand: in the Ramsey-based setting a valid termination argument typically must hold for the *transitive closure* of the program's transitions, rather than only for individual transitions. Thus, the proof of a termination argument's validity is much harder. In size-change [15] or variance analysis [3] the result is imprecision: the tools are fast but can only prove a limited set of programs due to inaccuracies in the underlying abstractions that facilitate reasoning about the transitive closure. In iterative-based approaches

⋆ Supported by the FWF grant S11403-N23 (RiSE) and the WWTF grant ICT12-059.

(*e.g.* [8], [14]) the result is lost performance and scalability, as symbolic model checking tools are ultimately used to reason about the program transition relation's transitive closure—something generally accepted as difficult.

In this paper we revisit the use of the Ramsey-based termination arguments used in the iterative-based approach to termination proving used in tools such as ARMC [23], Terminator [8], and the termination proving module of CProver [6,14]: rather than iteratively finding Ramsey-based termination arguments, we instead aim to iteratively find traditional lexicographic termination arguments. The advantage of this approach is that the validity checking step in the iterative process is much easier. The difficulty is that, outside of termination proving for rewrite systems, scalable methods for finding lexicographic ranking functions for whole programs are previously unknown.

We describe such a method. In our approach we keep information from all past failed proof attempts and use it to iteratively strengthen a lexicographic termination argument. Using experimental evidence we demonstrate dramatic performance improvements made possible by the new approach.

Related work. In this work we draw inspiration from the AProVE termination proving tool for rewrite systems [12], which proves termination of whole programs using what are effectively lexicographic arguments. The difficulty with AProVE, however, is that it has limited support for the discovery of supporting invariants. In our procedure we get the best of both worlds: lexicographic termination arguments are used, and invariants are found on demand via a reduction to tools for proving safety properties.

In our tool, during each iterative step of the proof search, we make use of constraint-based ranking function synthesis techniques from Bradley, Manna, and Sipma [4]. The difference here is that we iteratively enrich the termination argument using successful calls to a constraint-based tool on slices of the program, whereas constraint-based ranking function synthesis tools (*e.g.* [4], [5], [21], etc) were originally applied to entire programs.

Kroening *et al.* [14] optimize Ramsey-based iterative termination arguments using transitivity: attempts are made to strengthen a Ramsey-based termination argument such that it becomes a transitive relation, thus facilitating faster reasoning about the termination argument's validity. Note that in some simple cases the transitive and lexicographic arguments for a program can be similar, though lexicographic arguments are more strictly defined. The difference in our work is that we make use of all past failed termination proofs to find lexicographic termination arguments. Our choice results in increased time spent looking for termination arguments, but less time spent proving their validity.

Here we are addressing the performance of the iterative approach to termination proving, not techniques such as size-change or variance analysis. Fogarty and Vardi's experiments [10] indicate that Ramsey-based termination arguments are superior to lexicographic-based arguments in size-change.

Limitations. We are focusing primarily on arithmetic programs (*e.g.* programs that do not use the heap). In some cases we have soundly abstracted C programs

```
1    while x>0 and y>0 do
2         if * then
3             x := x - 1;
4         else
5             x = *;
6             y = y - 1;
7         fi
8    done
```

Fig. 1. Example terminating program. The symbol `*` is used to represent non-deterministic choice.

```
1    assume (x>0 and y>0);          1    assume (x>0 and y>0);
3    x := x - 1;                    5    x := *;
                                    6    y := y - 1;

          (Cycle 1)                          (Cycle 2)
```

Fig. 2. Two cycles in the program in Fig. 1

with heap to arithmetic programs (*e.g.* using a technique due to Magill *et al.* [16]); in other cases, as is standard in many tools (*e.g.* SLAM [2]), we essentially ignored the heap. Techniques that more accurately and efficiently reason about mixtures of heap and arithmetic are an area of open interest. Additionally, later in the paper we discuss some curious cases where linear lexicographic termination arguments alone are not powerful enough to prove termination, but linear Ramsey-based ones are. For these rare cases we describe some ad hoc strategies that facilitate the use of linear lexicographic termination arguments. In principle, however, if these approaches do not work, we would need to default to Ramsey-based arguments.

2 Example

Consider the example program in Fig. 1. When attempting to prove termination of this example the Terminator tool would, during its iterative process, end up examining two cycles in the program, as seen in Fig. 2. We know that the first cycle cannot be executed forever because `x` always decreases and is bound by 0. The second cycle also cannot be executed forever, as `y` always decreases and is bound by 0.

But what of paths that consist of a mixture of Cycle 1 and Cycle 2? To prove termination of any such path, we must verify that over any finite sequence (of any length) consisting of Cycle 1 and Cycle 2, at least one of `x` or `y` decreases and is bound by 0. If `oldx` and `oldy` are the values of `x` and `y` at the some previous position of the sequence, we must verify that at the end of the sequence:

$$(\mathtt{x} < \mathtt{oldx} \text{ and } 0 \leq \mathtt{oldx}) \text{ or } (\mathtt{y} < \mathtt{oldy} \text{ and } 0 \leq \mathtt{oldy}).$$

```
copied := 0;
while x>0 and y>0 do
    if copied=1 then
        assert((x<oldx and 0≤oldx) or (y<oldy and 0≤oldy));
    else if * then
        copied := 1;
        oldx := x;
        oldy := y;
    fi
    if * then
        x := x - 1;
    else
        x := *;
        y := y - 1;
    fi
done
```

Fig. 3. Termination argument validity check for the program in Fig. 1, with Ramsey-based termination argument ($\texttt{x} < \texttt{oldx}$ and $0 \leq \texttt{oldx}$) or ($\texttt{y} < \texttt{oldy}$ and $0 \leq \texttt{oldy}$). The instrumented code used by TERMINATOR's validity check is underlined.

Following Podelski & Rybalchenko's transition invariants [22], if we can find a finite set of ranking functions such that over any sub-sequence of transitions from one reachable program state to another, (*i.e.* over any pair of states in the transitive closure of the program's transitions), at least one of the ranking functions decreases and is bound by 0, then we have proved termination. We refer to this type of termination argument as a *Ramsey-based termination argument.*

To prove the validity of the termination argument discussed above, TERMINATOR would then use a known program transformation [8] to produce the program in Fig. 3. The `assert` command in this program fails iff the Ramsey-based termination argument is not valid. Model checking techniques for safety (*e.g.* SLAM [2], BLAST [13], CPROVER [6], IMPACT [18], WHALE [1], etc) can then be used to prove/disprove the `assert`. The problem with this strategy is that the safety proof is unnecessarily tricky: we need to prove that, after the `copied := 1` statement, *each* time the `assert` statement is reached it cannot fail. The safety prover is then effectively forced to find and prove an inductive transition invariant [22] that implies that the termination argument holds for every iteration of the loop after the assignment `copied := 1`. Experimentally we find that the performance of this strategy suffers dramatically as the complexity of the loop body increases. For simple programs (*e.g.* device drivers) with few nested loops, this approach suffices, but for more complex programs, problems arise.

In this paper we instead propose to use more sophisticated constraint techniques to find the *lexicographic termination argument* that, in this case, orders the ranking function `y` before `x`. In our notation from before we might express this argument as:

```
copied := 0;
while x>0 and y>0 do
    if copied=1 then
        assert((x<oldx and 0≤oldx and y≤oldy) or (y<oldy and 0≤oldy));
        exit();
    else if * then
        copied := 1;
        oldx := x;
        oldy := y;
    fi
    if * then
        x := x - 1;
    else
        x := *;
        y := y - 1;
    fi
done
```

Fig. 4. Termination argument validity check for lexicographic termination argument. The differences from Fig. 3 are underlined.

$$(\texttt{x} < \texttt{oldx} \text{ and } 0 \leq \texttt{oldx} \underline{\text{ and } \texttt{y} \leq \texttt{oldy}}) \text{ or } (\texttt{y} < \texttt{oldy} \text{ and } 0 \leq \texttt{oldy}).$$

Here, we require that either `y` decreases towards a bound, or `x` decreases towards a bound *and* `y` *does not increase*. To prove the validity of this termination argument we need only prove that this condition holds over any one cycle, rather than over any sequence of cycles. Therefore we call on a safety prover to prove that the `assert` found in Fig. 4 cannot fail.

The advantage of the problem in Fig. 4 over that in Fig. 3 is the call to `exit`(): we need only prove that the *first* call to the `assert` cannot fail, as only one call is possible. In many cases this change results in an enormous overall performance advantage, as no inductive transition invariant is required. The difficulty here is that we must use more powerful constraint-solving techniques to find the lexicographic termination argument. Experimentally we find that the increased time spent up-front looking for a stronger termination argument pays off in the end.

3 Procedure

In this section we describe our new lexicographic-based iterative termination proving procedure.

Programs, locations, paths. As usual (*e.g.* [17]) we assume that programs are represented as graphs with locations and edges labeled with transition relations. Here we represent the transitions between edges as commands with assignment or **assume** statements (from Nelson [20]). For example, the program in Fig. 1 would be represented as the graph in Fig. 5. In formulae describing sets of

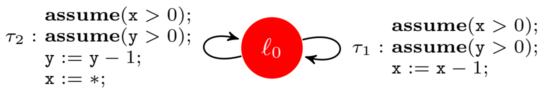


Fig. 5. Graph-based representation of the program from Fig. 1

states we can specify locations in the program's graph using the variable pc, which ranges over program locations ℓ_0, ℓ_1, etc. A *path* is a feasible sequence of transitions between states. A *cycle* is a path whose start and end states have the same program location ℓ, and that does not visit ℓ in between. We map from commands to relations on states in the usual way, *e.g.* $[\![\mathtt{x} := \mathtt{x}+1]\!] = \{(s,t) \mid t(\mathtt{x}) = s(\mathtt{x})+1 \wedge \forall v \in \textsc{Vars} \setminus \{\mathtt{x}\},\ t(v) = s(v)\}$. We map sequences of transitions to relations using relational composition, *e.g.* $[\![\langle\tau_1,\tau_2,\tau_3\rangle]\!] = [\![\tau_1]\!];[\![\tau_2]\!];[\![\tau_3]\!]$.

Termination arguments. A ranking function is a map from the state-space of the program to a well-ordered set. Ranking functions are used to measure the progress of the terminating process. A *linear* ranking function is of the form $r_1x_1 + \cdots + r_mx_m + r_{m+1}$ where $x_1, x_2, \ldots, x_m$ are the program variables. Our linear ranking functions range over the well-ordered set of the natural numbers with the relation $\leq$. Given a ranking function f, we define its *ranking relation* as

$$T_f = \{(s,t) \mid f(s) > f(t) \wedge f(s) \geq 0\}$$

i.e. all pairs of states over which f decreases and is bound by 0. Transitions in the ranking relation contribute to the progress of f. Similarly, we define a ranking function's *unaffecting relation* as

$$U_f = \{(s,t) \mid f(s) \geq f(t)\}$$

i.e. all pairs of states over which f is not increased. Transitions in the unaffecting relation do not impede the progress of f. Given a binary relation ρ over the state-space, we say that a ranking function f is *unaffected* by ρ if $\rho \subseteq U_f$.

We now consider $\Pi = \langle\rho_1,\rho_2,\ldots,\rho_n\rangle$, a finite sequence of n binary relations over the state-space, representing n cycles that are found during our iterative procedure. We define a *linear lexicographic ranking function* (LLRF) for Π as a finite sequence of n linear ranking functions $\langle f_1, f_2, \ldots, f_n\rangle$ such that $\forall i \in \{1,2,\ldots,n\}$: **a)** $\rho_i \subseteq T_{f_i}$, and **b)** $\forall j < i, \rho_i \subseteq U_{f_j}$. That is, f_i decreases and is bound by 0 over ρ_i, and $f_1, f_2, \ldots, f_{i-1}$ are all unaffected by ρ_i. Given a lexicographic ranking function, we can define the *lexicographic ranking relation* L as all pairs of states that, for some $i \in \{1,2,\ldots,n\}$, are contained within $U_{f_1} \cap U_{f_2} \cap \cdots \cap U_{f_{i-1}} \cap T_{f_i}$. Clearly $\bigcup \Pi \subseteq L$. Note that for any lexicographic ranking function, its lexicographic ranking relation is well-founded by construction. This is the reason why we need only verify that each individual transition obeys the lexicographic termination argument, rather than the transitive closure. In this paper termination arguments will take the form of lexicographic ranking relations.

Termination Procedure. Our iterative lexicographic-based termination proving procedure is found in Fig. 6. We begin with an empty termination argument, T.

```
input: program $P$

$T :=\ \emptyset$, empty termination argument
$\Pi :=\ \langle\rangle$, empty sequence of relations
Unused $:=\ \langle\rangle$, empty sequence of $(\Pi, T)$ pairs

repeat

  if $\exists$ cycle $\pi$ in $P$ s.t. $[\![\pi]\!] \not\subseteq T$ then
    let $n = \text{length}(\Pi) = \text{length}\langle\rho_1, \rho_2, \ldots, \rho_n\rangle$
    Successes $:=\ \emptyset$, empty set of $(\Pi, T)$ pairs

    for $i = 1$ to $n + 1$ do
      let $\Pi_i = \langle\rho_1, \rho_2, \ldots, \rho_{i-1}, [\![\pi]\!], \rho_i, \ldots, \rho_n\rangle$
      if $\exists$ a LLRF $F_i$ for $\Pi_i$
        let $L_i =$ the lexicographic ranking relation for $F_i$
        Successes $:= \{(\Pi_i, L_i)\} \cup$ Successes

    if |Successes| $\geq 1$
      randomly choose one $(\Pi_i, L_i) \in$ Successes and remove it
      $\Pi := \Pi_i$
      $T := L_i \supseteq \bigcup \Pi$
      Unused := (sequence of Successes) $\oplus$ Unused
    else
      if |Unused| $\geq 1$ then
        $(\Pi, T) :=$ head(Unused)
        Unused := Unused$\backslash\{(\Pi, T)\}$
      else
        report "Unknown"

  else
    report "Success"

end.
```

Fig. 6. Lexicographic-based iterative termination procedure. $\oplus$ denotes concatenation of finite sequences.

We search for a witness (a cycle π) to the failure of the validity of this argument. Our implementation of the search for a witness is an adaptation on the reduction to safety proving from Cook, Podelski, and Rybalchenko [8].

Our procedure then goes on to keep and use all of the witnesses (Π) to the failure of T. If there are none, we have proved termination. Otherwise if we find a witness, we add it to Π in the form of a relation. Each time a relation is added, a new LLRF is synthesized for Π. Each new termination argument T contains $\bigcup \Pi$, so we continue to add to Π until (hopefully) T is a valid termination argument for the program P. It is therefore useful to think of Π rather than T as representing the progress of the algorithm.

Once we have a sequence of relations $\Pi = \langle\rho_1, \rho_2, \ldots, \rho_n\rangle$, the LLRF for Π is synthesized by finding a linear ranking function f_i for each relation ρ_i in Π. We additionally attempt to satisfy the Unaffected constraints: That is, $\forall i \in \{1, 2, \ldots, n\}$, we require that ρ_i does not increase any of $f_1, f_2, \ldots, f_{i-1}$. We have then constructed a linear lexicographic ranking function $\langle f_1, f_2, \ldots, f_n\rangle$

```
1    while x>0 and y>0 and d>0 do
2      if * then
3        x := x - 1;
4        d := *;
5      else
6        x = *;
7        y = y - 1;
8        d = d - 1;
9      fi
10   done
```

(a)

```
1    while x>0 and y>0 and z>0 do
2      if * then
3        x := x-1;
4      else if * then
5        y := y-1;
6        z := *;
7      else
8        z := z-1;
9        x := *;
10     fi
11   done
```

(b)

Fig. 7. Example terminating programs

for Π. Previously known constraint-based techniques using Farkas' Lemma (*e.g.* [4],[5],[21]) are used to find the sequence of functions satisfying the above.

Note that for each new Π, we synthesize the LLRF anew, which allows each individual ranking function f for a particular relation ρ to change from one iteration to the next. This is necessary, as permanently designating a ranking function to each relation can lead to a failure to find a solution that does in fact exist. As an example, consider the loop in Fig. 7(a), which is the same as Fig. 1 except it features a decoy variable d. The lexicographic termination argument $\langle y, x\rangle$ we found earlier for Fig. 1 is clearly valid for this loop too. We examine two cycles: Lines 1,3,4, which induces $\rho_1 = [\![x > 0 \wedge y > 0 \wedge d > 0 \wedge x' = x-1 \wedge y' = y]\!]$; and Lines 1,6,7,8, which induces $\rho_2 = [\![x > 0 \wedge y > 0 \wedge d > 0 \wedge y' = y - 1 \wedge d' = d - 1]\!]$.

Suppose we find ρ_2 first, and choose $f_2 = d$ as its ranking function. Suppose we then find ρ_1. We need a LLRF for either $\langle \rho_1, \rho_2\rangle$ or for $\langle \rho_2, \rho_1\rangle$. If we require that $f_2 = d$ from the previous iteration, then this means we must find f_1 a linear ranking function for ρ_1 such that one of the two following options holds:

a) $\langle f_1, d\rangle$ is a LLRF for $\langle \rho_1, \rho_2\rangle$. So we need f_1 to be unaffected by ρ_2.
b) $\langle d, f_1\rangle$ is a LLRF for $\langle \rho_2, \rho_1\rangle$. So we need $f_2 = d$ to be unaffected by ρ_1.

Clearly **b)** is unsatisfiable because d isn't unaffected by ρ_1. **a)** is also unsatisfiable because to be a linear ranking function for ρ_1, f_1 must be of the form $r_x x + r_y y + c$ with $r_x > 0$, and therefore f_1 isn't unaffected by ρ_2. Therefore if we require the ranking function for ρ_2 to stay the same throughout the execution of our procedure, we may find no solutions, due to an earlier unlucky choice of ranking function. However, if we allow f_2 to be changed from d, we will be able to find our solution $\langle f_2, f_1\rangle = \langle y, x\rangle$, which is a valid lexicographic ranking function for $\langle \rho_2, \rho_1\rangle$, and for the whole loop.

Fortunately, synthesizing LLRFs for a small (and fixed order) Π is cheap, so the re-synthesis of the LLRFs has little effect on performance. This statement is not without a caveat: incremental approaches to safety proving in practice allow us to resume the validity checking from where we left off in the previous iteration, thus major changes to the ranking function can make for additional work in the safety prover. As a further optimization we could imagine using the interpolants found in the safety prover to help guide the search for even better termination arguments.

Choosing the lexicographic ordering. As mentioned previously, the relations in Π must be put in some lexicographic order $\langle \rho_1, \rho_2 \ldots \rho_n \rangle$ for a lexicographic ranking function to be found. As shown in Fig. 6, this is done by insertion — the relation that has just been found is inserted into the previous lexicographic ordering. This means that after the n^{th} relation is found, there are n places it can be inserted, *i.e.* n choices of ordering to consider. For each of the n orderings, we attempt to find a LLRF. If there are one or more orderings that yield solutions, we choose at random an ordering and its corresponding lexicographic ranking relation to form our new Π and T respectively.

The advantage of this method is that should we find that a certain ordering yields no solutions, we do not investigate it further. That is, if there does not exist a LLRF for some ordering Π, then there does not exist a LLRF for any ordering obtained by inserting relations into Π, and we do not investigate any such orderings. The disadvantage of this method is that it can be *too* selective, leading us to a dead end. We demonstrate this possibility in Fig. 7(b), then present our solution. We investigate three cycles: Lines 1,3, which induces $\rho_1 = [\![x > 0 \wedge y > 0 \wedge z > 0 \wedge x' = x - 1 \wedge y' = y \wedge z' = z]\!]$; Lines 1,5,6, which induces $\rho_2 = [\![x > 0 \wedge y > 0 \wedge z > 0 \wedge y' = y - 1 \wedge x' = x]\!]$; and Lines 1,8,9, which induces $\rho_3 = [\![x > 0 \wedge y > 0 \wedge z > 0 \wedge z' = z - 1 \wedge y' = y]\!]$. Suppose that during our procedure, the first two relations we find are ρ_1 and ρ_2. They have ranking functions $f_1 = x$ and $f_2 = y$ respectively. Note that ρ_1 does not increase y and ρ_2 does not increase x, so we may choose either $\langle \rho_1, \rho_2 \rangle$ or $\langle \rho_2, \rho_1 \rangle$ with LLRF $\langle x, y \rangle$ or $\langle y, x \rangle$ respectively.

– Suppose we choose $\langle \rho_2, \rho_1 \rangle$ with LLRF $\langle y, x \rangle$. Next we find ρ_3, and see that inserting it to form the new ordering $\langle \rho_2, \rho_3, \rho_1 \rangle$ yields a LLRF $\langle f_2, f_3, f_1 \rangle = \langle y, z, x \rangle$. This is a valid lexicographic ranking function for the whole loop, and so we have proved termination.
– Suppose we choose $\langle \rho_1, \rho_2 \rangle$ with LLRF $\langle x, y \rangle$. Next we find ρ_3, but there does not exist a LLRF for any one of $\langle \rho_3, \rho_1, \rho_2 \rangle$ or $\langle \rho_1, \rho_3, \rho_2 \rangle$ or $\langle \rho_1, \rho_2, \rho_3 \rangle$, so we have reached a dead end.

This example demonstrates that by investigating *only* the orderings obtained by inserting the new relation into the previous ordering, we may be unable to find an existing solution due to an earlier choice of ordering. Of course, we could investigate *all* possible permutations of the n relations to avoid this problem, but that strategy becomes infeasible once n becomes large [5], as on the n^{th} iteration we would need to investigate $n!$ cases rather than n.

In our solution (*i.e.* Fig. 6), in the event of more than one feasible ordering, we choose one randomly and keep the others in UNUSED, so that if a dead end is later reached, we may backtrack to the last random choice made, and investigate an alternative ordering. Cases such as the above that require the backtracking failsafe are uncommon. The insertion strategy with backtracking is fast because we only attempt to find n lexicographic ranking functions on the n^{th} iteration. The approach is robust because we will eventually investigate all lexicographic ranking functions we found, if necessary.

4 Towards Finding the *Right* Ranking Function

In many cases, there is more than one choice of Π that admits a LLRF, and for each Π, there may be more than one possible LLRF. Such cases give us the opportunity to consider which choices might be better than others, *i.e.* which termination argument is likely to be faster to validate using existing safety proving techniques. Note that in our setting the sequence Π affords us a great deal of information when trying to determine which argument to choose. In this section we describe several heuristics that we have found useful. We close this section with a discussion of some cases where no (linear) lexicographic termination argument exists, but linear Ramsey-based arguments can be found.

Shorter lexicographic ranking functions. Checking the validity of a lexicographic ranking function (as demonstrated in Fig. 4) becomes more difficult as the lexicographic ranking function becomes longer. This is because for a lexicographic ranking function of length n, we are checking, for each transition, whether any one of n conjunctive formulae hold.

We implemented an optimization that chooses a LLRF that uses the fewest unique ranking functions as possible. Then, if we have some of the f_i equal, we may eliminate the repeated ranking functions by keeping just the *first* occurrence of each unique ranking function. The resulting LLRF is shorter, and its lexicographic ranking relation contains $\bigcup \Pi$, so it forms our new termination argument. In one example from our experimental evaluation we found that proving termination was possible in 27s with this optimization turned on, and 157s without.

Unaffecting lexicographic ranking functions. Recall that a lexicographic ranking function $\langle f_1, f_2, \ldots, f_n \rangle$ for $\Pi = \langle \rho_1, \rho_2, \ldots, \rho_n \rangle$ must satisfy the Unaffecting constraints: every ρ_i must satisfy $\rho_i \subseteq U_{f_j} \forall j < i$. However we do not require $\rho_i \subseteq U_{f_j}$ for any $j > i$.

Intuitively, when attempting to prove the validity of a termination argument (which, ultimately, happens via the search for an inductive argument in the safety prover), it seems that checking the validity of a lexicographic ranking function is easier when the relations interfere minimally with the other relations' ranking functions, *i.e.* increase them as little as possible. That is, we wish to satisfy as many of the extra Unaffecting constraints $\{\rho_i \subseteq U_{f_j} \mid j > i\}$ as possible. This motivates the following definition.

```
while x<>0 do
    if x>0 then
        x := x-1;
    else
        x := x+1;
    fi
done
```

(a)

```
assume(m>0);
while x<>m do
    if x>m then
        x := 0;
    else
        x := x+1;
    fi
done
```

(b)

Fig. 8. Example programs where Ramsey-based linear termination arguments exist, but linear lexicographic termination arguments do not

Given a lexicographic ranking function $\langle f_1, f_2, \ldots, f_n\rangle$ for $\Pi = \langle \rho_1, \rho_2, \ldots, \rho_n\rangle$, its *Unaffecting Score* U is

$$U = \sum_{1\le i<j\le n} 1_{U_{f_j}}(\rho_i)$$

where the indicator function $1_{U_f}(\rho)$ equals 1 if $\rho \subseteq U_f$ and 0 otherwise. In other words, U is the number of extra unaffecting constraints satisfied. Note that we always have $0 \le U \le \frac{n(n-1)}{2}$, and requiring $U = 0$ is equivalent to the usual lexicographic ranking function constraints.

We implemented a constraint-based optimization that chooses a LLRF with highest possible Unaffecting Score. In our experiments the example mentioned above (that required 157s without optimizations) was proved terminating in 82s with this optimization turned on.

When linear lexicographic ranking relations are not enough. Existence of a linear Ramsey-based termination argument for a loop does not imply existence of a linear lexicographic termination argument for the same loop. We illustrate two simple but typical examples. See Fig. 8. For both examples we present a simple solution that alters the problem slightly, allowing us to continue to use lexicographic techniques to prove termination. Note that both of these simple workarounds are not new—variations upon these themes have been used in previous tools (*e.g.* APROVE [12]). Our intention here is to illustrate the type of problems that arise when moving from Ramsey-based to lexicographic termination arguments.

In Fig. 8(a), the variable `x` starts as any integer, then increases or decreases (as appropriate) until it equals 0, upon which the loop terminates. A valid Ramsey-based termination argument for the loop is:

$$(\texttt{x} < \texttt{oldx} \text{ and } 0 \le \texttt{oldx}) \text{ or } (\texttt{-x} < \texttt{-oldx} \text{ and } 0 \le \texttt{-oldx}).$$

However there does not exist a LLRF for the loop. Neither $\langle x, -x\rangle$ nor $\langle -x, x\rangle$ is valid, as every transition decreases one of the functions and increases the other. A solution to this problem is shown in Fig. 9(a). The variable `c` is introduced to

```
copied := 0;
c := 0;
while x<>0 do
   if copied=1 then
      assert(  (c=1 and x<oldx and 0≤oldx)
                                       or
               (c=2 and -x<-oldx and 0≤-oldx)
            );
      exit();
   else if * then
      copied := 1;
      oldx := x;
   fi
   if x>0 then
      if c=0 then c:=1;
      x := x-1;
   else
      if c=0 then c:=2;
      x := x+1;
   fi
done
```

(a)

```
copied := 0;
iters := 0;
assume(m>0);
while x<>m do
   if iters ≥ 1 then
      if copied=1 then
         assert(m-x<oldm-oldx and 0≤oldm-oldx);
         exit();
      else if * then
         copied := 1;
         oldx := x;
         oldm := m;
      fi
   fi
   if x>m then
      x := 0;
   else
      x := x+1;
   fi
   iters := iters+1;
done
```

(b)

Fig. 9. Modified validity check transformations for programs in Fig. 8. The modifications to the standard validity check are underlined.

record which of the two options was taken upon entry to the loop the first time through. In our procedure we instrument such variables into the representation of the program. Then, in the case where we cannot find a LLRF — and before resorting to a Ramsey-based termination argument — we would attempt to build the following lexicographic termination argument that case splits on `c`: $\langle f_1 \rangle = \langle x \rangle$ for `c=1` and $\langle f_2 \rangle = \langle -x \rangle$ for `c=2`. The relation would be encoded as

$$(\texttt{c} = 1 \text{ and } \texttt{x} < \texttt{oldx} \text{ and } 0 \leq \texttt{oldx}) \text{ or } (\texttt{c}{=}2 \text{ and } -\texttt{x} < -\texttt{oldx} \text{ and } 0 \leq -\texttt{oldx}).$$

This extension aims to deal with cases where there is a split-case at the beginning of the loop, necessitating seemingly conflicting ranking functions that prohibit construction of a lexicographic ranking function, but the loop is nonetheless terminating because the two cases are largely separate.

In Fig. 8(b), `m` and `x` start as any integers with `m` positive. If `x` is greater than `m`, `x` is set to zero. `x` is now less than `m`, so `x` increases until it equals `m`, upon which the loop terminates. A valid Ramsey-based termination argument for the loop is:

$$(\texttt{x} < \texttt{oldx} \text{ and } 0 \leq \texttt{oldx}) \text{ or } (\texttt{m-x} < \texttt{oldm-oldx} \text{ and } 0 \leq \texttt{oldm-oldx}).$$

However there does not exist a LLRF for the loop. Neither $\langle x, m - x \rangle$ nor $\langle m - x, x \rangle$ is valid, as every transition decreases one of the functions and increases the other. A simple solution to this problem is shown in Fig. 9(b). The variable `iters` records how many iterations of the loop have occurred. We then attempt to prove termination lexicographically by only checking transitions for which $\texttt{iters} \geq 1$, then $\texttt{iters} \geq 2$, $\texttt{iters} \geq 3$, etc. up to some finite limit at which point we give up. (Our failure to find a LLRF by the usual procedure means that we have already failed to prove termination for $\texttt{iters} \geq 0$). When examining the path found we can

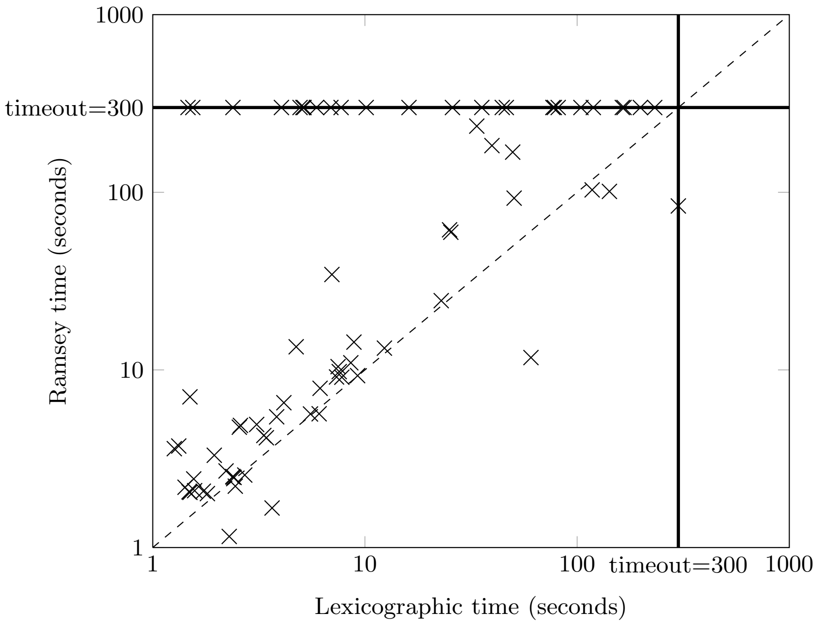


Fig. 10. Results of experimental evaluation comparing the lexicographic-based iterative termination prover from Fig. 6 to a re-implementation of TERMINATOR [8]. In total 390 termination benchmarks were used, with a timeout of 300s. Depicted here are the 82 cases in which radical differences in performance are seen (there are 42 cases where both tools timeout, and 266 easily solved cases) The lexicographic approach resulted in 26 fewer timeouts (*i.e.* the Ramsey-based termination procedure timed out on 68 benchmarks). The dotted line indicates equal performance of both methods. Note that on a log-log plot, results lying on a line parallel to the dotted line represent one method performing at a rate proportional to the other. Results were computed using an Intel 2.80Ghz processor running Windows 7. A source-code release of the tool and benchmarks is scheduled for 2013.

easily discover if the prefix of the cycle contributes to well-foundedness using an extra constraint check. In our example, we need only attempt to prove for $\texttt{iters} \geq 1$ (shown in Fig. 8(b)) to find that the lexicographic ranking function $\langle f_2 \rangle = \langle m - x \rangle$ is valid. This extension aims to deal with loops which include an initialization procedure that occurs over the first few iterations (if at all), necessitating ranking functions that conflict with those needed for the main termination argument. It allows us to construct lexicographic termination arguments that do not need to take into account the first few iterations of the loop.

5 Experimental Evaluation

To evaluate our approach we have implemented the algorithm from Fig. 6 as an option in the T2 termination proving tool[1]. The underlying safety prover used to

[1] A source-code based release of this tool together with benchmarks is scheduled for release in 2013.

check termination argument validity in T2 is a re-implementation of Impact [18]. We then applied the tool to a set of 390 termination proving benchmarks, drawn from a variety of applications (*e.g.* device drivers, the Apache webserver, Postgres SQL server, integer approximations of numerical programs from a book on numerical recipes [24], integer approximations of benchmarks from LLBMC [19], etc). Note that, as we mentioned earlier, in some cases we have soundly abstracted C programs with data-structures to pure arithmetic programs using a technique due to Magill *et al.* [16]. In other cases we have ignored the heap. We have used the same input files for all experiments and configurations, thus the treatment of heap is orthogonal to the investigation here.

To see the difference between Ramsey-based and lexicographic-based iterative termination proving, we compared our new procedure to T2's re-implementation of the original Terminator procedure (which includes an integration of the optimization from Kroening *et. al* [14]). We ran the two variants of T2 on the 390 termination benchmarks, with a timeout of 300s. See Fig. 10 for the results (in logarithmic scale). Here we have excluded 266 cases where both tools were able to prove/disprove termination in under 3 seconds, as well as 42 cases where both tools timed out. The remaining 82 cases are shown in the figure. The most dramatic aspect of the results is the decrease in timeouts: 26.

6 Conclusion

In this paper we have reconsidered the form of termination argument used in iterative-based termination proving [8]: rather than iteratively finding Ramsey-based termination arguments, we have instead developed a method that iteratively finds traditional lexicographic termination arguments. This approach has some disadvantages (*i.e.* more complex ranking function synthesis) and advantages (*i.e.* easier termination argument validity checking). Overall the experimental evidence indicates that the advantages outweigh the disadvantages.

References

1. 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)
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., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.W.: Variance analyses from invariance analyses. In: POPL (2007)
4. 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)
5. 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)

6. Clarke, E., Kroning, 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)
7. Codish, M., Genaim, S., Bruynooghe, M., Gallagher, J., Vanhoof, W.: One loop at a time. In: WST (2003)
8. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
9. Dershowitz, N., Lindenstrauss, N., Sagiv, Y., Serebrenik, A.: A general framework for automatic termination analysis of logic programs. Communication and Computing 12(1/2) (2001)
10. Fogarty, S., Vardi, M.Y.: Büchi Complementation and Size-Change Termination. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 16–30. Springer, Heidelberg (2009)
11. Geser, A.: Relative termination. Doctoral dissertation, University of Passau (1999)
12. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-Safety Proofs for Systems Code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
14. 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)
15. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL (2001)
16. 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)
17. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: Safety (1995)
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. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 146–161. Springer, Heidelberg (2012)
20. Nelson, G.: A generalization of Dijkstra's calculus. TOPLAS 11(4) (1989)
21. 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)
22. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS (2004)
23. 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 (2007)
24. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing (1989)
25. Turing, A.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines (1949)

Structural Counter Abstraction

Kshitij Bansal[1], Eric Koskinen[1,⋆], Thomas Wies[1], and Damien Zufferey[2,⋆⋆]

[1] New York University
[2] IST Austria

Abstract. Depth-Bounded Systems form an expressive class of well-structured transition systems. They can model a wide range of concurrent infinite-state systems including those with dynamic thread creation, dynamically changing communication topology, and complex shared heap structures. We present the first method to automatically prove fair termination of depth-bounded systems. Our method uses a numerical abstraction of the system, which we obtain by systematically augmenting an over-approximation of the system's reachable states with a finite set of counters. This numerical abstraction can be analyzed with existing termination provers. What makes our approach unique is the way in which it exploits the well-structuredness of the analyzed system. We have implemented our work in a prototype tool and used it to automatically prove liveness properties of complex concurrent systems, including nonblocking algorithms such as Treiber's stack and several distributed processes. Many of these examples are beyond the scope of termination analyses that are based on traditional counter abstractions.

1 Introduction

Graph transformation systems [9] are a well-studied formalism for describing concurrent computations. A *depth-bounded system* [17, 26] is a graph transformation system for which there exists a bound on the length of all simple (i.e. acyclic) paths in all reachable graphs. Depth-bounded systems are also well-structured transition systems (WSTS) [10]. This makes them an attractive target for automated analysis because there are generic algorithms for deciding a number of verification problems for WSTS [1].

Depth-bounded systems are also among the most expressive classes of WSTS, subsuming *e.g.* Petri nets and their monotonic extensions [18]. They can model a wide range of concurrent systems including those with dynamic thread creation, dynamically changing communication topology, and complex shared heap data structures. Many concurrent systems are depth-bounded. For instance, Actor-style message passing systems often fall into this class. Other systems have natural depth-bounded abstractions that preserve important properties. For example, consider the lock-free stack due to Treiber [25] (see Figure 1), which uses atomic *compare-and-swap* instructions to implement nonblocking stack operations. This algorithm can be abstracted to a depth-bounded system by ignoring the order of the elements in the stack. This abstraction

⋆ Supported in part by the CMACS NSF Expeditions in Computing award 0926166.

⋆⋆ Supported in part by the European Research Council (ERC) Advanced Investigator Grant QUAREM and by the Austrian Science Fund (FWF) project S11402-N23.

preserves the termination/progress behavior of the algorithm. Similar depth-bounded abstractions can be obtained for a wide variety of concurrent algorithms.

In this paper, we present the first method to automatically prove fair termination of depth-bounded systems. We focus on a notion of *weak fairness* that is consistent with the *finite delay property* for Petri nets [5]. However, our technique also extends to other fairness conditions. Many liveness properties of practical interest (including progress guarantees: wait-, lock-, and obstruction-freedom) are reducible to termination under weak fairness. The problem is difficult; it subsumes the structural termination problem for transfer nets (i.e. termination for all possible input markings), which is undecidable [16]. Despite this difficulty, we show that one can build on existing verification techniques for WSTS to obtain an approximate analysis for this problem that is both practical and sufficiently precise to prove fair termination of complex systems.

The key technical contribution of this paper is a method that automatically constructs a precise numerical abstraction of a depth-bounded system from a precomputed inductive invariant of the system. The inductive invariant is assumed to be given as a finite set of nested graphs in which nested subgraphs can be unfolded arbitrarily often. Thus, each nested graph is a symbolic representation of the (infinite) set of concrete graphs obtained by such unfoldings. We associate a counter with each of the nested subgraphs, tracking how often it can be unfolded. From these augmented nested graphs we then compute a numerical transition system that simulates the depth-bounded system. This so-called *structural counter abstraction* can then be analyzed using existing termination provers. The number and meaning of counters in the numerical abstraction is not fixed *a priori* but, instead, depends on the structure of the reachable configuration graphs (described by the inductive invariant). Our method thus provides a more precise alternative to traditional counter abstractions [3,7,21] for concurrent systems.

The benefit of our approach is that it can utilize existing reachability analyses for depth-bounded systems to obtain the inductive invariant [27], and existing termination analyses for numerical programs [6,22]. We have implemented our method in a prototype tool and applied it to prove liveness properties of various concurrent systems, including nonblocking algorithms such as Treiber's stack, as well as distributed processes. These systems are beyond the scope of traditional counter abstraction techniques.

Contributions. We present the first automatic technique for proving fair termination of depth-bounded systems. Our technique enables the automated verification of liveness properties for a large class of concurrent infinite-state systems. What makes our approach unique is the way in which it exploits the monotonicity of the system. Our algorithmic technique of computing a numerical abstraction from an inductive invariant, introduced in this paper, promises applications beyond liveness properties. For instance, the same technique can be used to strengthen an inductive invariant of a depth-bounded system with numerical constraints, enabling proofs of complex safety properties.

2 Overview

Motivating example. Consider Treiber's stack [25], a non-blocking algorithm, given in the C-like code in Fig. 1. The algorithm implements a stack with a simple linked-list. The two operations, `push` and `pop` use the *compare-and-swap* (`CAS`) instruction

```
struct node {
   struct node *next;
   value t data;
};

struct stack {
  struct node *Top;
};
struct stack *S;

void push(value t v) {
  struct node *t, *x;
  x = alloc();
  x→data = v;
  do { t = S→Top; x→next = t; }
  while (¬CAS(&S→Top,t,x));
}

void init() {
   S = alloc();
   S→Top = NULL;
}

value t pop() {
  struct node *t, *x;
  do {
    t = S→Top;
    if (t == NULL) return EMPTY;
    x = t→next;
  } while (¬CAS(&S→Top,t,x));
  return t→data;
}
```

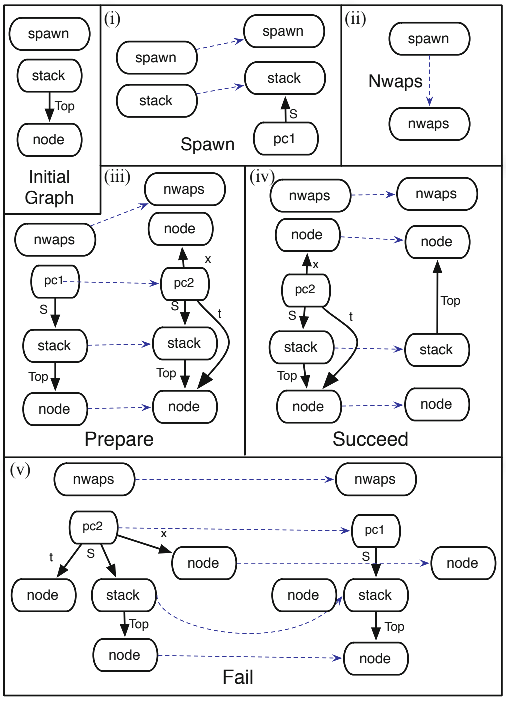


Fig. 1. Source code of Treiber's stack [25] and its abstraction as a graph transformation system

to atomically modify a location in memory. `CAS(l,v,v')` atomically examines the value at location l and, if it is equivalent to v, sets l to value v'. In this section, we will describe how we are able to prove lock-freedom of this algorithm via a reduction to fair termination of a depth-bounded system.

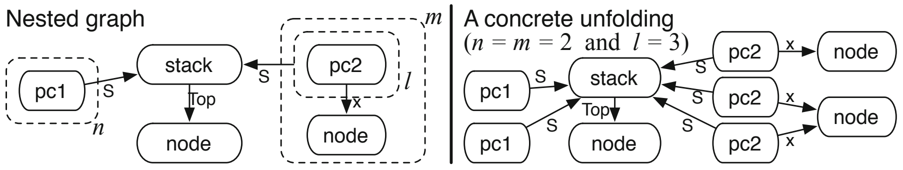
We can represent Treiber's stack algorithm as a depth-bounded system, by abstracting over the values and order of the elements in the stack. In the depth-bounded abstraction of Treiber's stack, the graphs represent the state of the heap, *i.e.*, the linked list implementing the stack, and thread objects describing the local states of all clients currently executing push and pop operations. The abstraction is obtained from the concrete transition system of Treiber's stack by ignoring the values of `next` pointers connecting the vertices in the linked list of the stack. In this abstraction, there may still be unboundedly many elements in the stack as well as unboundedly many clients operating on the stack. However, since the list vertices are no longer connected, they can no longer form simple paths of arbitrary length in the heap graph. At this level of abstraction, `push` and `pop` become indistinguishable. Both operations have the same control-flow structure: they iteratively read the top of the stack and attempt to modify it until the `CAS` operation succeeds. The actual modification of the stack is non-deterministic in both operations.

Depth-bounded abstractions of programs can be computed automatically from the program's source code using shape analysis techniques. These techniques are orthogonal to the contribution of this paper. In Fig. 1 we give the graph rewriting system for the depth-bounded abstraction of Treiber's stack. The initial state is a graph consisting of the vertex spawn, indicating that clients can be spawned, and the stack and its Top element which is some node. There are five rewrite rules. (i) The Spawn rule replaces a stack vertex with an identical stack vertex that is connected to a new vertex pc1 representing a client in an initial thread state before the `CAS` (pc1 refers to its owning stack via edge S). The dotted line indicates how the left-hand-side of the rule is replaced by the right-hand-side: the stack vertex on the left is replaced with the stack vertex on the right. (ii) Spawning may cease when the Nwaps rule is applied. Here, the spawn vertex is replaced with a nwaps vertex. The effect is that both the Spawn and Nwaps rules are disabled, but the remaining rules now become enabled. (iii) In the Prepare rule, a client reads the stack's Top pointer and prepares a new element (pointed to by x) to be pushed or popped onto the stack. There are then two cases that correspond to whether or not the `CAS` operation succeeds (depending on whether the local pointer t agrees with Top). (iv) In the Succeed case, the stack is updated to point to the new element and the old element is disregarded. This is a generalization that encompasses both push and pop. (v) Alternatively, the `CAS` may fail, as given by the Fail case. The stack is unchanged and the client forgets what it read and retries.

We can prove that Treiber's stack is lock-free by showing that its depth-bounded abstraction always terminates modulo a weak fairness constraint. The fairness constraint is that the Nwaps rule cannot be continuously enabled without being applied, i.e., a fair run of the system will only spawn finitely many clients. It does not matter whether we allow process spawning only in an initial phase (as in our model), or at any time.

The key contribution of this paper is a technique that automatically constructs a precise numerical abstraction of a depth-bounded system from a given inductive invariant of the system. We refer to this numerical abstraction as the *structural counter abstraction*. The structural counter abstraction then enables us to prove weakly fair termination of the system. Our approach utilizes existing reachability analyses for well-structured transition systems to obtain the inductive invariant, and existing termination analyses for numerical programs to prove termination of the structural counter abstraction. In the following, we explain the construction of the counter abstraction for Treiber's stack.

Nested graphs. Above we saw that graph rewrite rules transform a subcomponent of a concrete graph into another concrete graph. However, we will need to work with (potentially infinitely many) instances of graph subcomponents. So we instead work with *nested graphs* (formal definitions provided in Section 5) in which subcomponents are given counters that indicate an upper bound on how many times they may be duplicated. For Treiber's stack, consider this abstract graph on the left hand side:



The set of concrete graphs represented by this nested graph are those in which the dotted subcomponents are repeated some number of times but at most as many times as determined by the associated counter. For instance, the left dotted subgraph is repeated at most n times. A component may itself contain nested sub-components. An example of an unfolded concrete graph is given on the right hand side. Notice that the pc2 vertices occur at different frequencies per node vertex. Also note that counters always refer to the *total* number of copies of their component. This representation can be thought of as a more precise alternative to counter abstractions [3,7,21], in that we associate counters with nested graph components rather than merely program locations. We say that a nested graph $\widehat{G}_1$ is *covered* by nested graph $\widehat{G}_2$ if the set of concrete graphs obtainable from unfoldings of $\widehat{G}_2$ is contained within the set of concrete graphs obtainable from unfoldings of $\widehat{G}_1$. Determining whether $\widehat{G}_2$ covers $\widehat{G}_1$ is decidable and, as we will see, helps ensure that the structural counter abstraction can be effectively computed.

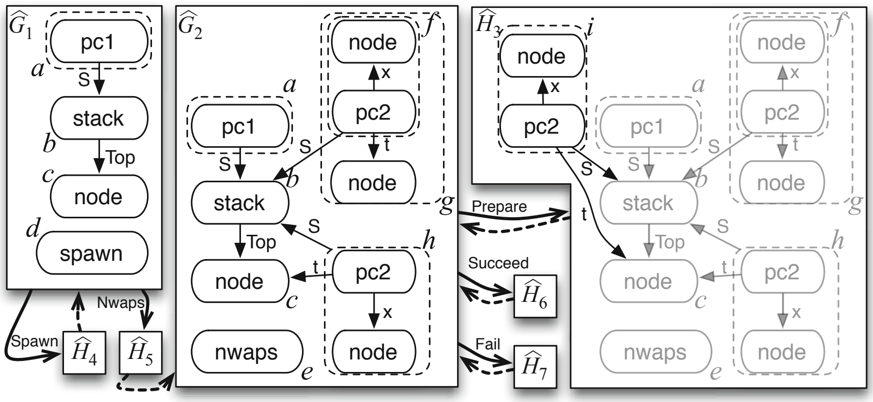


Fig. 2. Structural counter abstraction for Treiber's stack. Numerical transition constraints are omitted for readability. Here the inductive invariant is given by nested graphs $\widehat{G}_1$ and $\widehat{G}_2$.

Obtaining the structural counter abstraction. We begin with a nested graph representation of the inductive invariant. For Treiber's stack the invariant is $\widehat{G}_1$ and $\widehat{G}_2$ in Fig. 2. This invariant (obtained, *e.g.*, via [27]) is a finite set of nested graphs and is an over-approximation of the reachable states of the system. $\widehat{G}_1$ describes states in which spawning may still occur (indicated with a spawn vertex) and $\widehat{G}_2$ describes states in which spawning has ceased (indicated with a nwaps vertex) and arbitrarily many clients have performed Prepare, Suceed or Fail.

We begin to construct the structural counter abstraction by associating a counter variable with each subcomponent of each nested graph in the inductive invariant. For example in Fig. 2, we have established counter variables a, b, c, d with components of $\widehat{G}_1$ and additional counter variables e, f, g, h in $\widehat{G}_2$. In our generation of the structural counter abstraction, we leverage the fact that the invariant is *closed* under rewrite rules. That is, whenever we apply a rewrite rule to a nested graph $\widehat{G}$ in the inductive invariant, we obtain $\widehat{H}$ that is already covered by some other nested graph $\widehat{G}'$ in the invariant.

To construct the abstraction, we apply each rewrite rule, one at a time, for every possible match in one of the nested graphs in the invariant. For example, in Fig. 2 we can apply the Prepare rule as follows. We first unfold one instance of the pc1 vertex a in $\widehat{G}_2$, obtaining a separate pc1 vertex to which we apply the Prepare rule. This produces a new nested graph $\widehat{H_3}$ that extends $\widehat{G_2}$ with a new subgraph. We add a new counter variable i for this new subgraph in $\widehat{H_3}$. Notice that, because the inductive invariant is maximal, $\widehat{H_3}$ is covered by the existing graph $\widehat{G_2}$ (hence the dotted edge from $\widehat{H_3}$ to $\widehat{G_2}$). It is covered because the isomorphic subgraphs with associated counters i and h in $\widehat{H_3}$ can both be represented by the subgraph with associated counter h in $\widehat{H_3}$. From the point of view of the concrete graph transformation system, we can think of this covering edge as an ϵ-transition: every rewrite rule that is susequently applied to $\widehat{H_3}$ can also be applied to $\widehat{G_2}$. The structural counter abstraction is a numerical transition system that reflects the corresponding changes to the counter values when rewrite and covering edges between nested graphs are taken. There are several other possible instances where rules can be applied to this inductive invariant. (These involve graphs $\widehat{H_4}$, $\widehat{H_5}$, $\widehat{H_6}$, and $\widehat{H_7}$ which have been omitted for lack of space.) For example, one can apply the Spawn rule in $\widehat{G}_1$ and obtain $\widehat{H_4}$ which has two pc1 subgraphs. This new graph $\widehat{H_4}$ is, again, covered by $\widehat{G}_1$ and the two pc1 subgraphs can be merged into the pc1 subgraph in $\widehat{G}_1$.

Structural counter abstraction. The structural counter abstraction is represented as a simple control-flow graph program $\mathcal{N} = (\mathit{Locs}, s_0, \mathit{Vars}, \Delta)$. Here, Locs refers to the control locations. There is one location per nested graph in the inductive invariant, respectively, per nested graph obtained by application of a rewriting rule. The variables Vars are the structural counters in the nested graphs, and Δ is a set of commands that change the counter values according to the rewriting and covering steps. s_0 is the initial state. An excerpt of the structural counter abstraction for Treiber's stack that captures parts of Fig. 2 is as follows:

$$\mathcal{N} \equiv (\{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7\}, s_0, \{a, b, c, \dots\}, \{(\ell_2, \delta_{23}, \ell_3), (\ell_3, \delta_{32}, \ell_2), \dots\}) \text{ where}$$
$$s_0 \equiv (\ell_1, \{b \mapsto 1, c \mapsto 1, d \mapsto 1, \_ \mapsto 0\})$$
$$\delta_{23} \equiv a' = a - 1 \wedge i' = i + 1 \wedge Id|_{\{a,i\}} \quad \delta_{32} \equiv h' = h + i \wedge i' = 0 \wedge Id|_{\{h,i\}}$$

$Id|_S$ is the identity mapping on the variables, excluding those in S. The transition constraint δ_{23} captures the application of the Prepare rule on $\widehat{G_2}$ yielding $\widehat{H_3}$. The transition constraint δ_{32} captures the covering transition from $\widehat{H_3}$ back to $\widehat{G_2}$. The initial state s_0 encodes the initial graph of the system which consists of one spawn, one stack, and one node vertex. The fairness constraints on the original system can be translated to fairness constraints on the structural counter abstraction in a straightforward manner. The structural counter abstraction we produce is then fit to be analyzed by an existing termination analysis tool such as Terminator [6] or ARMC [22].

Prototype. In Section 6 we describe our prototype tool that automates all steps required to prove fair termination of depth-bounded systems: generation of the inductive invariant, construction of the structural counter abstraction, and the final termination proof. It is able to prove fair termination of the Treiber stack model in less than 10 seconds. A simple counter abstraction that distinguishes only between processes at different control locations would yield a system with fair infinite traces. It is crucial to distinguish between the processes at location pc2 that may still succeed and those that are bound to fail. This is achieved by our more fine-grained structural counter abstraction.

3 Background

Posets and wqos. A *quasi-ordering* $\leq$ is a reflexive and transitive relation $\leq$ on a set X. In the following $X(\leq)$ is a quasi-ordered set. The *downward closure* of $Y \subseteq X$ is $\downarrow Y = \{ x \in X \mid \exists y \in Y. x \leq y \}$, Y is *downward-closed* if $Y = \downarrow Y$. An *upper bound* $x \in X$ of a set $Y \subseteq X$ is such that for all $y \in Y$, $y \leq x$. A nonempty set $D \subseteq X$ is *directed* if any two elements in D have a common upper bound in D. A set $I \subseteq X$ is an *ideal* of X if I is downward-closed and directed. A quasi-ordering $\leq$ on a set X is a *well-quasi-ordering* (wqo) if any infinite sequence $x_0, x_1, x_2, \ldots$ of elements from X contains an increasing pair $x_i \leq x_j$ with $i < j$.

(Well-Structured) Labeled Transition Systems. A *(labeled) transition system* is a tuple $\mathcal{T} = (S, s_0, \mathit{Act}, \longrightarrow)$ where S is a set of states, $s_0 \in S$ an initial state, Act a set of action labels, and $\longrightarrow \subseteq S \times \mathit{Act} \times S$ is a transition relation. We define $s \xrightarrow{a} s'$ iff $(s, a, s') \in \longrightarrow$. For $A \subseteq \mathit{Act}$, we define $s \xrightarrow{A} s'$ iff $s \xrightarrow{a} s'$ for some $a \in A$. We further define the *post operator* for an action a as $\mathsf{post}_a : \mathcal{P}(S) \to \mathcal{P}(S)$ with $\mathsf{post}_a(X) = \{ x' \in S \mid \exists x \in X. x \xrightarrow{a} x' \}$ and extend it to $\mathsf{post}_{\mathcal{T}}$ by $\mathsf{post}_{\mathcal{T}}(X) = \bigcup_{a \in \mathit{Act}} \mathsf{post}_a(X)$. The *reachability set* of a transition system $\mathcal{T}$, denoted $\mathrm{Reach}(\mathcal{T})$, is defined by $\mathrm{Reach}(\mathcal{T}) = \mathrm{lfp}^{\subseteq}(\lambda X.\{s_0\} \cup \mathsf{post}_{\mathcal{T}}(X))$. A set $X \subseteq S$ is called an *invariant* of $\mathcal{T}$ if $\mathrm{Reach}(\mathcal{T}) \subseteq X$, and X is called *inductive* if $\mathsf{post}_{\mathcal{T}}(X) \subseteq X$. A *well-structured transition system* (WSTS) is a tuple $\mathcal{T} = (S, s_0, \mathit{Act}, \to, \leq)$ where $(S, s_0, \mathit{Act}, \to)$ is a transition system and $\leq \subseteq S \times S$ a wqo that is *monotonic* with respect to $\to$, i.e., for all s_1, s_2, t_1, a such that $s_1 \leq t_1$ and $s_1 \xrightarrow{a} s_2$, there exists t_2 such that $t_1 \xrightarrow{a} t_2$ and $s_2 \leq t_2$. The *covering set* of a well-structured transition system $\mathcal{T}$, denoted $\mathrm{Cover}(\mathcal{T})$, is defined by $\mathrm{Cover}(\mathcal{T}) = \downarrow \mathrm{Reach}(\mathcal{T})$.

Graphs. We use a standard notation for (directed) graphs, denoted as tuples of the form (V, E), with $E \subseteq V \times V$. We define *(vertex) labeled graphs* over a set of labels VL as graphs with labels for each vertex and denote them as (V, E, ν) where $\nu : V \to \mathit{VL}$ is the *vertex-labeling function*. For the rest of the paper we fix VL, a finite set of labels and we denote by Graphs the set of all labeled graphs with labels VL. Also, unless explicitly stated otherwise, whenever we say graph, we refer to a labeled graph. We use the standard notions of (partial) homomorphisms, isomorphisms, subgraphs, etc. For a set $V' \subseteq V$ of vertices of a graph $G = (V, E)$, we denote by $G[V'] = (V', E \cap V' \times V')$ the subgraph *induced* by V'. We further denote by $\preceq$ the quasi-ordering induced by subgraph isomorphisms, i.e., $G \preceq H$ iff G is isomorphic to a subgraph of H. We write $G \cong H$ if G and H are isomorphic.

Graph Transformation Systems. We use an adaptation of the standard notion of graph transformation systems with the single pushout approach [9] to labeled directed graphs. A *rewriting rule* is a partial morphism $r : G_L \rightharpoonup G_R$, where G_L is called *left-hand side* and G_R is called *right-hand side*. A *match* of r is a total injective morphism $m : G_L \to G$. Given a rule r and a match $m : G_L \to G$, a *rewriting step* is the *pushout* of r and

m, which consists of a graph H and two graph morphisms $r' : G \rightharpoonup H, m' : G_R \rightarrow H$ such that $m' \circ r = r' \circ m$ and for every pair of morphisms $r'' : G \rightharpoonup H'$ and $m'' : G_R \rightharpoonup H'$ there exists a unique morphism $f : H \rightharpoonup H'$ with $f \circ m' = m''$ and $f \circ r' = r''$. It is known that pushouts are guaranteed to exist, that they are unique up to isomorphism and that they can be effectively constructed. A *graph transformation system* (GTS) $\mathcal{R}$ is a tuple (R, G_0), where R is a set of rewriting rules and G_0 an initial graph. A GTS $\mathcal{R} = (R, G_0)$ induces a transition system $\mathcal{T}(\mathcal{R}) = (\mathit{Graphs}, G_0, R, \xrightarrow{R})$ where R is a finite set of rewriting rules, and $\xrightarrow{R}$ is the union of all relations $\xrightarrow{r}$, for $r \in R$. The subgraph ordering $\preceq$ is monotonic with respect to graph rewriting.

Lemma 1. *Let $\mathcal{R} = (R, G_0)$ be a GTS, then $\preceq$ is monotonic with respect to $\xrightarrow{R}$.*

4 Weakly Fair Termination of Depth-Bounded Systems

In this section, we formally define the class of systems that we consider in this paper and the type of questions that we answer about these systems.

The *depth* of a graph G is the length of the longest simple path in the undirected version of G, obtained by taking the symmetric closure of the edges. For $k \in \mathbb{N}$, we denote by $\mathcal{G}_{\leq k}$ the set of all graphs with depth at most k. We say that a set of graphs $\mathcal{G}$ is *depth-bounded* if $\mathcal{G} \subseteq \mathcal{G}_{\leq k}$ for some $k \in \mathbb{N}$. A *depth-bounded system* (DBS) is a GTS $\mathcal{R} = (R, G_0)$, whose reachable configuration graphs are depth-bounded, i.e., $\mathrm{Reach}(\mathcal{T}(\mathcal{R})) \subseteq \mathcal{G}_{\leq k}$, for some $k \in \mathbb{N}$. We call k a *bound* of the system. From [26, Proposition 12] it follows that $\preceq$ is a wqo on depth-bounded sets of graphs.

Lemma 2. *For any $k \in \mathbb{N}$, $(\mathcal{G}_{\leq k}, \preceq)$ is a wqo.*

Thus, Lemmas 1 and 2 imply that depth-bounded GTSs induce WSTSs.

Theorem 3. *Let $\mathcal{R} = (R, G_0)$ be a DBS, then $(\mathrm{Cover}(\mathcal{R}), G_0, R, \xrightarrow{R}, \preceq)$ is a WSTS.*

Let $\mathcal{T} = (S, s_0, \mathit{Act}, \rightarrow)$ be a transition system. A *finite trace* π of $\mathcal{T}$ is a sequence $s_0\, a_0\, s_1\, a_1 \ldots a_{n-1}\, s_n$, with $s_i \in S$ and $a_i \in \mathit{Act}$ such that $s_i \xrightarrow{a_i} s_{i+1}$, for all $0 \leq i < n$; we define infinite traces $s_0\, a_0\, s_1\, a_1 \ldots$ correspondingly. We say that an action $a \in \mathit{Act}$ is *enabled* in a state s, if there exists a state s' such that $s \xrightarrow{a} s'$. Let $\mathcal{F} = \{A_0, \ldots, A_m\}$ be a set of subsets of Act. An infinite trace $s_0\, a_0\, s_1\, a_1 \ldots$ is *weakly fair* with respect to $\mathcal{F}$ if for every A_j, $0 \leq j \leq m$, there are infinitely many i such that $a_i \in A_j$ or there are infinitely many i such that no action in A_j is enabled in s_i.

Definition 4. *Given a transition system $\mathcal{T}$ and a finite set $\mathcal{F}$ of sets of actions of $\mathcal{T}$, the* weakly fair non-termination problem *asks whether there exists an infinite trace π of $\mathcal{T}$ such that π is weakly fair with respect to $\mathcal{F}$. We refer to the complementary problem as the* weakly fair termination problem *(WFT).*

Theorem 5. *Weakly fair termination is undecidable for depth-bounded systems.*

The proof of Theorem 5 goes by reduction of the structural termination problem for transfer nets to WFT of transfer nets. The former problem is known to be undecidable [16]. Transfer nets are subsumed by depth-bounded systems.

5 Structural Counter Abstraction

We now see the formal treatment of how one obtains the structural abstraction of a given depth-bounded system and how it is used to give approximate answers to the weakly fair termination problem. For the remainder of this section, let $\mathcal{R}$ be a depth-bounded system. We systematically construct the structural counter abstraction of $\mathcal{R}$ from an inductive invariant of $\mathcal{R}$. However, we are not interested in arbitrary inductive invariants but in those that are downward-closed with respect to graph embedding. Since graph embedding is a wqo on depth-bounded graphs, such downward-closed sets are finite unions of ideals of the embedding order [27]. Each ideal can itself be finitely represented and we can compute symbolically the effect of transition on this representation. This enables us to compute a form of closure on the inductive invariant that yields the structural counter abstraction. We start by formalizing this representation of ideals.

Nested graphs. We represent downward-closed depth-bounded sets of graphs as finite sets of *nested graphs*. Formally, a *nested graph* $\widehat{G}$ is a tuple (V, E, ν, l) where (V, E, ν) is a labeled graph and $l : V \to \mathbb{N}$ maps each vertex to its *nesting level*. We abuse notation and denote the labeled graph of a nested graph $\widehat{G}$ by G. We extend the notion of homomorphism to nested graphs as expected, i.e., homomorphisms on nested graphs also preserve the nesting levels of vertices.

Meaning of nested graphs. Intuitively, a nested graph $\widehat{G}$ represents the set of *concrete* graphs that can be obtained by recursively unfolding the nested subgraphs of $\widehat{G}$ arbitrarily often. In the following, we make these notions formal.

We define a *one-step unfolding* relation on nested graphs $\widehat{G} = (V, E, \nu, l)$ and $\widehat{H} = (V', E', \nu', l')$, denoted $\widehat{G} \rightsquigarrow \widehat{H}$, as follows. For $i \geq 1$, denote all vertices at nesting level i or higher by $V_{\geq i} = \{ v \in V \mid l(v) \geq i \}$. Unfolding involves *duplicating* the subgraph induced by $V_{\geq i}$ and reducing the nesting level of all vertices in the copy of $V_{\geq i}$ by one. Formally, we have $\widehat{G} \rightsquigarrow \widehat{H}$ iff for some $i \geq 1$ there exists a partition U, W_1, W_2 of V' and a homomorphism $h : H \to G$ such that $H[U \cup W_1] \cong G \cong H[U \cup W_2]$, $H[W_1] \cong G[V_{\geq i}] \cong H[W_2]$ under (natural restrictions of) h, $W_1 \times W_2 \cap E' = \emptyset$, for all $v' \in V' \setminus W_2$, $l'(v') = l(h(v'))$, and for all $v' \in W_2$, $l'(v') = l(h(v')) - 1$.

We then define the concretization $\gamma(\widehat{G})$ of a nested graph $\widehat{G}$ as the downward closure (with respect to the embedding order) of the set of all unfoldings of $\widehat{G}$: $\gamma(\widehat{G}) = {\downarrow}\{ H \mid \widehat{G} \rightsquigarrow^* \widehat{H} \}$. We extend γ to sets of nested graphs $\widehat{\mathcal{G}}$ as expected: $\gamma(\widehat{\mathcal{G}}) = \bigcup_{\widehat{G} \in \widehat{\mathcal{G}}} \gamma(\widehat{G})$.

Inclusion of Nested Graphs. We next show that inclusion on nested graphs is decidable. Let $\widehat{G} = (V, E, \nu, l)$ and $\widehat{H} = (V', E', \nu', l')$ be nested graphs. Define the relation $\sqsubseteq$ on nested graphs as $\widehat{G} \sqsubseteq \widehat{H}$ iff $\gamma(\widehat{G}) \subseteq \gamma(\widehat{H})$. An *inclusion mapping* for $\widehat{G}$ and $\widehat{H}$ is a homomorphism $\widehat{h} : (V, E, \nu) \to (V', E', \nu')$ satisfying the following additional properties: (i) for all $v \in V$, $l(v) \leq l'(\widehat{h}(v))$; (ii) $\widehat{h}$ is injective with respect to level 0 vertices in V': for all $v, w \in V$, $v' \in V'$, $\widehat{h}(v) = \widehat{h}(w) = v'$ and $l'(v') = 0$ implies $v = w$; (iii) for all distinct $u, v, w \in V$ such that $\widehat{h}(u) = \widehat{h}(v)$, and u and v are both neighbors of w, $l(u) > l(w)$ and $l(v) > l(w)$.

Theorem 6. *Let $\widehat{G}$ and $\widehat{H}$ be nested graphs. Then $\widehat{G} \sqsubseteq \widehat{H}$ iff there exists an inclusion mapping $\widehat{h} : \widehat{G} \to \widehat{H}$. The problem of deciding the existence of $\widehat{h}$ is NP-complete.*

To see that the problem is in NP, note that each of the conditions for inclusion mapping can be checked in polynomial time. NP-hardness follows from the fact that the problem subsumes the subgraph isomorphism problem.

Nested graph rewriting. We lift application of rewrite rules to nested graphs by using inclusion mappings as the notion of a *match*. Intuitively, inclusion mappings allow us to apply the rewrite rule to an unfolding of the graph that contains the left-hand-side of the rule as a subgraph. Formally, we extend the notion of pushout to nested graphs in a natural way by using the homomorphisms defined on nested graphs. For a rewriting rule $r : G_L \to G_R$, naturally lift the notion and define $\widehat{r} : \widehat{G_L} \to \widehat{G_R}$. A *match* of $\widehat{r}$ is an inclusion mapping $\widehat{m} : \widehat{G_L} \to \widehat{G}$.

Lemma 7. *Given a rule $\widehat{r} : \widehat{G_L} \to \widehat{G_R}$ and a match $\widehat{m} : \widehat{G_L} \to \widehat{G}$, there exists a nested graph $\widehat{G}'$ and an injective inclusion mapping $\widehat{h} : \widehat{G_L} \to \widehat{G}'$ such that $\widehat{G} \rightsquigarrow^* \widehat{G}'$. Moreover, $\widehat{G}'$ and $\widehat{h}$ can be constructed in polynomial time.*

Let $\widehat{G}'$ be the nested graph and $\widehat{h} : \widehat{G_L} \to \widehat{G}'$ the injective inclusion mapping, as described in Lemma 7. Then there exists a pushout $\widehat{r}' : \widehat{G}' \rightharpoonup \widehat{H}, \widehat{h}' : \widehat{G_R} \to \widehat{H}$ for $\widehat{r}$ and $\widehat{h}$. This pushout defines a *rewriting step of nested graphs* $\widehat{G} \xrightarrow{\widehat{r}} \widehat{H}$.

Constructing the structural counter abstraction. In the following, we assume that $\widehat{\mathcal{I}}$ is a finite set of nested graphs such that $\gamma(\widehat{\mathcal{I}})$ is a downward-closed inductive invariant of $\mathcal{R}$. From $\widehat{\mathcal{I}}$ we then construct the structural counter abstraction. The precision of this abstraction depends on the precision of $\widehat{\mathcal{I}}$. The most precise downward-closed inductive invariant of $\mathcal{R}$ is the covering set $\mathrm{Cover}(\mathcal{T}(\mathcal{R}))$. Unfortunately, this set is in general not computable for depth-bounded systems[1], even though the covering problem[2] is decidable [26]. However, we can employ existing algorithms [27] that compute downward-closed inductive approximations of the covering set. In practice, these algorithms often yield precisely $\mathrm{Cover}(\mathcal{T}(\mathcal{R}))$. This is confirmed by our experiments in Section 6. In fact, we did not encounter a significant precision loss in any of our examples.

Let G_0 be the initial graph of $\mathcal{R}$ and let $\widehat{G_0}$ be the nested graph obtained by equipping G_0 with a nesting level function mapping all nodes to 0. Further, let R be the set of rewriting rules of $\mathcal{R}$. We define a set of *rewriting edges* E_R as follows: $E_R = \{ (\widehat{G}, r, \widehat{H}) \mid \widehat{G} \in \widehat{\mathcal{I}}, r \in R, \widehat{H} \in \widehat{\mathcal{G}}, \widehat{G} \xrightarrow{\widehat{r}} \widehat{H} \}$. That is, E_R describes the set of one step rule applications on the nested graphs in the inductive invariant. The set E_R is finite up to isomorphism of nested graphs. Next, define the set $\widehat{\mathcal{J}} = \{\widehat{G_0}\} \cup \{ \widehat{H} \mid (\widehat{G}, r, \widehat{H}) \in E_R \}$. From the fact that $\widehat{\mathcal{I}}$ is an inductive invariant it follows that, for all $\widehat{H} \in \widehat{\mathcal{J}}$ there exists $\widehat{G} \in \widehat{\mathcal{I}}$ such that $\widehat{H} \sqsubseteq \widehat{G}$. Fix one such $\widehat{G}$ for each $\widehat{H} \in \widehat{\mathcal{J}}$ and let E_C be the set of all pairs $(\widehat{H}, \widehat{G})$. We call the elements of E_C *covering edges*. Let $\mathcal{E} = E_R \cup E_C$. In Fig 2, we saw this construction for the example of Treiber's stack starting with an inductive invariant. The solid edges between nested graphs correspond to rewrite edges and the dashed ones to covering edges. At the end of Section 2, we also saw an excerpt of the counter abstraction, next we describe how this is done in general.

[1] This follows from the undecidability of place-boundedness of transfer nets [8].

[2] The covering problem for DBS asks whether for given a $\mathcal{R}$ and graph G, $G \in \mathrm{Cover}(\mathcal{T}(\mathcal{R}))$.

The abstraction is a tuple $\mathcal{N} = (\mathit{Locs}, s_0, \mathit{Vars}, \Delta)$ where $\mathit{Locs} = \{\, \ell_{\widehat{G}} \mid \widehat{G} \in \widehat{\mathcal{I}} \cup \mathcal{J} \,\}$ is a set of control locations, $\mathit{Vars} = \{\, x_v \mid v \in V(\widehat{G}), \widehat{G} \in \mathcal{I} \cup \mathcal{J} \,\}$ is a set of counter variables , one for each vertex of a nested graph in $\mathcal{I} \cup \mathcal{J}$, and $\Delta = \{\, \delta_e \mid e \in \mathcal{E} \,\}$ is a set of commands, one for each edge in $\mathcal{E}$. The command δ_e associated with an edge $e = (\widehat{G}, \widehat{H})$ is of the form $(\ell_{\widehat{G}}, \rho_e, \ell_{\widehat{H}})$ where ρ_e is a *transition constraint* over primed and unprimed versions of the variables in Vars. The initial state of $\mathcal{N}$ is $s_0 = (\ell_{\widehat{G_0}}, \eta_0)$ where η_0 is a function mapping counters to natural numbers and defined as $\eta_0(x_v) = 1$ if $v \in V(\widehat{G_0})$, and 0 otherwise. Further, let $\sigma_{\mathcal{R}} : \Delta \rightharpoonup R$ be a partial mapping defined as $\sigma_{\mathcal{R}}(\delta_e) = r$ if e is a rewriting edge for rule r.

The definition of the transition constraint δ_e for an edge $e \in \mathcal{E}$ depends on whether e is a rewriting or a covering edge. We first consider the case that e is a rewriting edge $(\widehat{G}, r, \widehat{H})$. In order to perform a rewrite (which only transforms level-0 vertices) we need to unfold the graph $\widehat{G}$. As mentioned in Lemma 7, this can be done efficiently giving us $\widehat{G} \rightsquigarrow^* K$. Each unfolding step gives a homomorphism, which can be composed together to give $h : K \to \widehat{G}$. Further, from the pushout we get a partial homomorphism $r' : K \rightharpoonup \widehat{H}$. Let V be the vertices of $\widehat{G}$, U the vertices of K, and W the vertices of $\widehat{H}$. Further, let U_0 be the level-0 vertices of K and define $\overline{U_0} = U \setminus U_0$. Similarly, let W_0 be the level-0 vertices of $\widehat{H}$. Then, the transition constraint ρ_e for e is given by the conjunction of the following constraints:

$$x_v = \sum_{u \in h^{-1}(v) \cap \overline{U_0}} x'_{r'(u)} + \left| h^{-1}(v) \cap U_0 \right|, \quad \text{for all } v \in V \tag{1}$$

$$x'_w = 1, \quad \text{for all } w \in W_0 \tag{2}$$

$$y' = 0, \quad \text{for all } y \in \mathit{Vars} \setminus \{\, x_w \mid w \in W \,\} \tag{3}$$

During unfolding of $\widehat{G}$ to $\widehat{H}$, if some vertex v with count x_v is duplicated, then constraint (1) ensures that all counts for the duplicates sum up to x_v. Level-0 vertices get a special treatment, since they may be transformed by the rewrite rule. Similarly, (2) takes care of level-0 vertices in the rewritten graph. The constraint (3) encodes that only counters of vertices associated with the successor location have non-zero values.

For covering edges $e = (\widehat{H}, \widehat{G})$, we use the inclusion mapping $\widehat{h} : \widehat{H} \to \widehat{G}$ between the two nested graphs to define the transition constraint δ_e. Let W be the vertices of $\widehat{G}$, W_0 the level-0 vertices of $\widehat{G}$, and V the vertices of $\widehat{H}$. The inclusion mapping encodes which vertices $v \in V$ are collapsed to a single vertex $w \in W$, yielding the constraint

$$x'_w = \sum_{v \in \widehat{h}^{-1}(w)} x_v, \quad \text{for all } w \in W \tag{4}$$

Then δ_e is the conjunction of constraint (4) and constraints (2) and (3), which are the same as in the case of a rewriting edge.

Finally, the fairness constraints $\mathcal{F}_{\mathcal{R}}$ of $\mathcal{R}$ can be translated to fairness constraints $\mathcal{F}_{\mathcal{N}}$ of $\mathcal{N}$ using the partial function $\sigma_{\mathcal{R}}$ as follows: $\mathcal{F}_{\mathcal{N}} = \{\, \sigma_{\mathcal{R}}^{-1}(R_i) \mid R_i \in \mathcal{F}_{\mathcal{R}} \,\}$.

The numerical abstraction induces a transition system $\mathcal{T}(\mathcal{N}) = (S, s_0, \Delta, \xrightarrow{\Delta})$ with states $S = \mathit{Locs} \times \mathbb{N}^{\mathit{Vars}}$, i.e., a program location along with an evaluation of the counters. The transition relation $\xrightarrow{\Delta}$ is as expected. The details of the following soundness theorem may be found in the technical report [2].

Theorem 8 (Soundness). *If* $(\mathcal{T}(\mathcal{R}), \mathcal{F}_{\mathcal{R}})$ *has a weakly fair infinite trace, then so does* $(\mathcal{T}(\mathcal{N}), \mathcal{F}_{\mathcal{N}})$.

6 Evaluation

We implemented a prototype of our algorithm as an extension to the PICASSO [20,27] tool. PICASSO takes as input a depth-bounded systems and computes a so called *abstract coverability tree* (ACT). The nodes of the ACT are nested graphs and its construction is similar to the Karp-Miller tree for Petri nets. The maximal nodes in the ACT form a downward-closed inductive invariant, $\widehat{\mathcal{I}}$, of the input system. From this invariant we generate a structural counter abstraction, $\mathcal{N}$, that is optimized and then analyzed with the ARMC [22] termination prover.

A naive implementation of the method described in Section 5 produced structural counter abstractions that were too big for current termination provers. For instance, for Treiber's stack, having one variable for each vertex of each nested graph in the inductive invariant and those obtained by applying rewrite rules led to an abstraction with over 170 variables and 40 transitions. We therefore optimized the generation of the abstraction to get a smaller counter program with the same termination properties. When we generate the constraints for a transition, we decompose the transition into three steps: unfolding, morphism, and covering. These steps lead to many intermediate locations and transitions. We eliminate the intermediate steps by using the quantifier elimination procedure for linear integer arithmetic in PRINCESS [24]. We collect the constraints generated for each step and quantify away the variables at the intermediate locations. The resulting constraint describes a single transition with the same source and target locations as the original three-step transition, using only the variables at those locations. Furthermore, we observed that in many places constant values are assigned to the variables because they represent nodes on nesting level 0. We propagate the constant values using a combination of lightweight abstract interpretation and constraint propagation. We use an abstract domain that maps the variables to $\mathbb{N} \cup \bot$. A variable v is mapped to a value n in $\mathbb{N}$ when we can deduce that v is always equal to n, otherwise v is mapped to $\bot$.From the abstract fixed point we extract variable/value pairs and eliminate the variables by replacing them with their associated values. Lastly, instead of using one variable per node and graph, we reuse the variables across different graphs. The renaming is done by finding a minimal coloring of a graph where the nodes are variables and there is an edge between two nodes if the corresponding variables are used at the same location.For Treiber's stack, we reduced the abstraction to 6 variables and 4 transitions.

Transition predicates. We observed that ARMC finds easily the predicates that involve one or two variables, but not the predicates requiring more variables. Fortunately, ARMC can take transition predicates as part of its input. We manually give hints to PICASSO in the form of variables names, usually corresponding to control-states. Those names are turned into transition predicates by summing the variables. For example, in the numerical abstraction of Treiber's stack we specified a simple predicate indicating that the sum of all the process counters was either unchanged or decreasing.

Table 1. Experimental results. The columns show the number of locations, variables, and transitions in the counter abstraction, and the running times, in seconds, for computing the inductive invariant, constructing the abstraction, and for proving termination.

Example	#loc	#v	#t	$\widehat{\mathcal{I}}$	$\mathcal{N}$	ARMC	**Total**
Split/merge	4	3	9	1.5	6.8	0.1	8.4
Work stealing, 3 processors	4	4	20	1.7	13.1	0.2	15.0
Work stealing, parameterized	2	3	4	1.5	5.6	0.1	6.2
Compute server job queue	2	5	4	1.6	6.1	0.1	7.8
Chat room	5	34	80	9.8	61.3	5 min	6 min
Map reduce	6	10	15	2.0	8.8	0.2	11.0
Map reduce with failure	6	15	21	2.3	11.1	0.9	14.3
Treiber's stack (coarse-grained)	2	6	4	1.9	7.2	0.2	9.3
Treiber's stack (fine-grained)	3	14	13	2.7	14.2	1.2	17.1
Herlihy/Wing queue	3	16	25	3.8	24.9	6.5	34.2
Michael/Scott queue (dequeue only)	4	7	23	2.8	13.0	0.6	16.4
Michael/Scott queue (enqueue only)	7	15	53	3.8	43.7	7.6	55.1
Michael/Scott queue	9	31	224	25.0	265.0	3 wks	3 wks

Results. Table 1 summarizes the results of our experiments. Our implementation is parallelized and ran on a server using 26 cores. Memory consumption was not an issue. We examined a collection of depth-bounded transition systems, including distributed processes and concurrent algorithms. The examples and the tool can be downloaded from the PICASSO web site [20]. We applied our method to prove global progress properties of those systems. Fairness is used to limit the number of clients, requests, and failures. Details about the encoding of fairness constraints can be found in the technical report [2]. Our experiments show that our approach can quickly prove termination of complex systems. The structural counter abstraction is concise and maintains the necessary information in order to prove termination.

The split/merge example is a parallel computation where a master sends jobs to a pool of workers. We also proved termination of (non-)parameterized versions of a work stealing algorithm. From [13] we considered systems obtained from Scala implementations of a chat room and a map reduce algorithm (with and without failure). As shared memory examples, we considered the model of Treiber's stack [25] described in Section 2 as well as a more fine-grained variant with push and pop modeled independently. We analyzed a model of the Herlihy/Wing concurrent queue [14] which requires an additional fairness constraint to ensure that dequeue operations cannot execute without enqueue operations ever taking steps. This is needed because the dequeue operation retries if the queue is empty. Finally, we also considered the Michael/Scott queue [19] where the order between the elements is abstracted. This example results in an abstraction that is very large for today's termination provers. We therefore also show the results for simpler models where enqueue and dequeue operations are considered in isolation.

7 Related Work

Depth-bounded systems (DBS) were first introduced by Meyer in [17] as a fragment of the π-calculus. In his paper, he showed that DBS are well-structured and that termination (without fairness constraints) is decidable. Termination without fairness has only limited practical applications because the initial state of the system is fixed. With a fixed initial state one cannot model systems with an infinite set of reachable states without losing termination, since we only consider finitely branching systems.

The idea of using reachability analyses to obtain numerical abstractions of programs whose states can be described by graphs is by itself not new. In particular, such techniques have been studied for proving safety and liveness properties of heap manipulating programs [4,12,23]. Our technique differs substantially from these approaches in the way the numerical abstraction is computed. Specifically, our technique is based on *ideal abstractions* [27] for computing over-approximations of the covering sets of WSTS and it exploits the monotonicity of the analyzed system, i.e., that the behavior observable from a given graph is subsumed by the behavior observable from any larger graph. Finally, the abstract domain of nested graphs can model unbounded recursive unfolding structures that naturally occur in complex concurrent systems and that are difficult to capture using traditional shape analysis domains.

Joshi and König study graph transformation systems that are well-structured with respect to the graph minor ordering [15]. Our approach targets a different application domain. We consider rewriting rules with injective matching. Systems with this semantics are not monotonic with respect to graph minors and therefore not well-structured under this ordering. On the other hand, the graph minor ordering is a wqo for arbitrary graphs, while the subgraph ordering is a wqo only for graphs bounded in the length of their simple paths. The two approaches thus consider orthogonal classes of WSTS.

An application of our results is proving nonblocking properties of concurrent algorithms. Others have considered approaches directly targeted on this goal. Gotsman *et al.* [11] describe a thread-modular proof technique. While their work enables thread-local reasoning, it is only suitable in instances where there are simple environmental invariants (*i.e.* other threads do not execute certain actions infinitely often).

8 Conclusion

We have shown a novel technique for proving fair termination of algorithms described as depth-bounded systems. Despite the fact that this problem is undecidable, we showed that one can build on existing verification techniques to obtain an approximate analysis that is both practical and sufficiently precise to prove fair termination of complex concurrent systems such as Treiber's stack. We have shown that our method is sound, and demonstrated viability with a prototype implementation.

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. Bansal, K., Koskinen, E., Wies, T., Zufferey, D.: Structural counter abstraction. Technical Report TR2012-947, New York University (2012)

3. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic Counter Abstraction for Concurrent Software. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 64–78. Springer, Heidelberg (2009)
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. Carstensen, H.: Decidability Questions for Fairness in Petri Nets. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 396–407. Springer, Heidelberg (1987)
6. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
7. Delzanno, G., Raskin, J.-F., Van Begin, L.: Towards the Automated Verification of Multithreaded Java Programs. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 173–187. Springer, Heidelberg (2002)
8. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset Nets Between Decidability and Undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)
9. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Handbook of graph grammars and computing by graph transformation, pp. 247–312. World Scientific Publishing Co., Inc. (1997)
10. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
11. Gotsman, A., Cook, B., Parkinson, M.J., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: POPL. ACM (2009)
12. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: POPL, pp. 239–251. ACM (2009)
13. Haller, P., Sommers, F.: Actors in Scala. Artima (January 2012)
14. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
15. Joshi, S., König, B.: Applying the Graph Minor Theorem to the Verification of Graph Transformation Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 214–226. Springer, Heidelberg (2008)
16. Mayr, R.: Undecidable problems in unreliable computations. Theor. Comput. Sci. 297(1-3), 337–354 (2003)
17. Meyer, R.: On Boundedness in Depth in the π-Calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Fifth IFIP International Conference on Theoretical Computer Science–TCS 2008. IFIP, vol. 273, pp. 477–489. Springer, Boston (2008)
18. Meyer, R., Gorrieri, R.: On the Relationship between π-Calculus and Finite Place/Transition Petri Nets. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 463–480. Springer, Heidelberg (2009)
19. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC (1996)
20. Picasso, http://pub.ist.ac.at/~zufferey/picasso/termination
21. 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)
22. 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 (2007)
23. 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)

24. Rümmer, P.: A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008)
25. Treiber, R.: Systems programming: Coping with parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center (1986)
26. Wies, T., Zufferey, D., Henzinger, T.A.: Forward Analysis of Depth-Bounded Processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)
27. Zufferey, D., Wies, T., Henzinger, T.A.: Ideal Abstractions for Well-Structured Transition Systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 445–460. Springer, Heidelberg (2012)

Extending Quantifier Elimination to Linear Inequalities on Bit-Vectors

Ajith K. John[1] and Supratik Chakraborty[2]

[1] Homi Bhabha National Institute, BARC, Mumbai, India
[2] Dept. of Computer Sc. & Engg., IIT Bombay, India

Abstract. We present an algorithm for existentially quantifying variables from conjunctions of linear modular equalities (LMEs), disequalities (LMDs) and inequalities (LMIs). We use sound but relatively less complete and cheaper heuristics first, and expensive but more complete techniques are used only when required. Our experiments demonstrate that our algorithm outperforms alternative quantifier elimination techniques based on bit-blasting and Omega Test. We also extend this algorithm to work with Boolean combinations of LMEs, LMDs and LMIs.

1 Introduction

Existential quantifier elimination (henceforth called QE) is the process of transforming a formula containing existential quantifiers into a semantically equivalent quantifier-free formula. This has a number of important applications in formal verification and program analysis, such as computing abstractions of symbolic transition relations, computing strongest postconditions of program statements, computing predicate abstractions and generating code fragments by automatic program synthesis techniques.

Verification and analysis tools often assume unbounded data types like `integer` or `real` for program variables. QE techniques for unbounded data types [4,8] are therefore often used in program analysis, verification and synthesis. However, a program executing on a machine with fixed-width words really uses fixed-width bit-vector operations. It is known [2,12] that program analysis assuming unbounded data types may not be sound if the implementation uses fixed-width words, and if overflows are not detected and accounted for. This motivates us to investigate QE techniques for constraints involving fixed-width words. Specifically, we present techniques for QE from Boolean combinations of linear modular (bit-vector) equalities, disequalities and inequalities.

Let p be a positive integer constant, $x_1, \ldots, x_n$ be p-bit non-negative integer variables, and $a_0, \ldots, a_n$ be integer constants in $\{0, \ldots, 2^p - 1\}$. A *linear term* over $x_1, \ldots, x_n$ is a term of the form $a_1 \cdot x_1 + \cdots a_n \cdot x_n + a_0$. A Linear Modular Equality (henceforth called LME) is a formula of the form $t_1 = t_2 \pmod{2^p}$, where t_1 and t_2 are linear terms over $x_1, \ldots, x_n$. Similarly, a Linear Modular Disequality (henceforth called LMD) is a formula of the form $t_1 \neq t_2 \pmod{2^p}$, and a Linear Modular Inequality (henceforth called LMI) is a formula of the form $t_1 \bowtie t_2$

(mod 2^p), where $\bowtie \in \{<, \leq\}$. For brevity, we will use "LMC" (for Linear Modular Constraint) when the distinction between LME, LMD and LMI is not important. In the LMCs given above, 2^p is conventionally called the modulus of the LMC. Since every variable in an LMC with modulus 2^p represents a p-bit integer, we will assume without loss of generality that whenever we consider a conjunction of LMCs sharing a variable, all the LMCs have the same modulus.

In our earlier work [1], we had presented a QE algorithm for Boolean combinations of LMEs and LMDs that is efficient in practice. Unfortunately, techniques for dealing with LMIs involve significantly more technicalities than those for dealing with LMEs and LMDs, and require development of more sophisticated techniques. This paper presents results of our investigations in this direction.

Earlier Work: Efficient procedures for reasoning about LMEs and LMDs were discussed in [1,10,11,12]. Bjørner et al [2] showed that the satisfiability problem for conjunctions of difference logic constraints in modular arithmetic is NP-complete. Their work also demonstrated that several intuitive equivalences that hold for inequalities over reals and integers do not necessarily hold for LMIs. QE from a conjunction of LMCs can be achieved by bit-blasting [3], followed by bit-level QE. However this technique irretrievably destroys the word-level structure of the problem, and scales poorly as the width of bit-vectors increases. A QE problem for a conjunction of LMCs can also be presented as a QE problem for a conjunction of inequalities in Integer Linear Arithmetic (ILA) and congruences [7]. Alternatively, each LMC can be reduced to a set of ILA constraints [3], and QE techniques for ILA, such as Omega Test [8], can be used to eliminate integers corresponding to specified bit-vectors. Unfortunately, these techniques have been found to scale poorly in practice [3]. In addition, recovering word-level constraints from the results is often difficult, especially when several variables are quantified. In this paper, we present an alternative approach that tries to overcome most of these drawbacks in practice.

2 QE from a Conjunction of LMCs

Let A denote a conjunction of LMCs over variables $x_1, \ldots, x_n$. We wish to compute a Boolean combination of LMCs, say φ, such that $\varphi \equiv \exists x_1 \cdots \exists x_t. A$. Let us initially focus on the simpler problem of existentially quantifying a single variable from a conjunction of LMCs. For clarity of exposition, we use x to denote the variable to be quantified

Notation and Preliminaries: To simplify notation, we assume that all LMCs in the remainder of the paper have modulus 2^p for some positive integer p, unless stated otherwise. We use letters x, y, z, x_1, $x_2, \ldots$ to denote variables, use a, a_1, $a_2, \ldots$, b, b_1, $b_2, \ldots$ to denote constants, and use s, s_1, $s_2, \ldots$, t, t_1, $t_2, \ldots$ to denote linear terms. The letters d, d_1, $d_2, \ldots$ are used to denote LMDs, l, l_1, $l_2, \ldots$ are used to denote LMIs, and c, c_1, $c_2, \ldots$ are used to denote LMCs. Furthermore, we use D, D_1, $D_2, \ldots$ to denote conjunctions of LMDs, I, I_1, $I_2, \ldots$ to denote conjunctions of LMIs, and C, C_1, $C_2, \ldots$, A, A_1, $A_2, \ldots$ to denote conjunctions of

LMCs. For a linear term t, we use $-t$ to denote the additive inverse of t modulo 2^p.

Since $(t_1 < t_2)$ is semantically equivalent to both $(t_2 \geq 1) \wedge (t_1 \leq t_2 - 1)$ and $(t_1 \leq 2^p - 2) \wedge (t_1 + 1 \leq t_2)$, there is no loss of generality in assuming that LMIs are restricted to be of the form $t_1 \leq t_2$. However, for clarity of exposition, we allow LMIs of the form $t_1 < t_2$, whenever convenient. An LME or LMD $t_1 \bowtie t_2$, where $\bowtie \in \{=, \neq\}$, can be equivalently expressed as $2^\mu \cdot x \bowtie t$, where t is a linear term free of x, and μ is an integer such that $0 \leq \mu \leq p$ (see [1]). Note that this does not sacrifice generality since we can set μ to p if the LME/LMD is free of x.

For every linear term t_1 and variable x, we define $\kappa(x, t_1)$ to be an integer in $\{0, \ldots, p\}$ such that t_1 is equivalent to $2^{\kappa(x,t_1)} \cdot e \cdot x + t$, where t is a linear term free of x, and e is an odd number in $\{1, \ldots, 2^p - 1\}$. Note that if t_1 is free of x, then $\kappa(x, t_1) = p$. The definition of $\kappa(x, \cdot)$ can be extended to (conjunctions of) LMCs as follows. Let c be an LME/LMD equivalent to $2^\mu \cdot x \bowtie t$, where $\bowtie \in \{=, \neq\}$ and t is free of x. We define $\kappa(x, c)$ to be μ in this case. If t_1, t_2 are linear terms, then $\kappa(x, t_1 \leq t_2)$ is defined to be $min(\kappa(x, t_1), \kappa(x, t_2))$. Finally, if $c_1, \ldots, c_m$ are LMCs, then $\kappa(x, \bigwedge_{i=1}^{m} (c_i))$ is defined to be $\min_{i=1}^{m}(\kappa(x, c_i))$. Observe that if C is a conjunction of (possibly one) LMCs and if $\kappa(x, C) = k$, then only the least significant $p - k$ bits of x affect the satisfaction of C. We will say that x is in the support of C if $\kappa(x, C) < p$.

Lemma 1. *Let A be a conjunction of LMCs containing at least one LME. Let $2^{k_1} \cdot x = t_1$ be the LME with the minimum $\kappa(x, \cdot)$ value among the LMEs in A. Then $\exists x.\, A \equiv C_1 \wedge \exists x.\, C_2$, where C_1 is a conjunction of LMCs free of x, and C_2 is a conjunction of $2^{k_1} \cdot x = t_1$ and (possibly zero) LMIs and LMDs, each of which has $\kappa(x, \cdot)$ less than k_1.*

We omit the proof of this and other lemmas due to space constraints. The reader is referred to [14] for all proofs.

Example: All LMCs in this example have modulus 8. Consider the problem of computing $\exists y.\, ((2^1 y = 5x + 2) \wedge (2^0 y \neq 6x + 7z) \wedge (2^0 \cdot 5y + z \leq 2^1 y) \wedge (2^1 \cdot 3y \leq x + z))$. This can be equivalently expressed as $\exists y.\, ((2y = 5x + 2) \wedge (y \neq 6x + 7z) \wedge (5y + z \leq 5x + 2) \wedge (3 \cdot (5x + 2) \leq x + z))$. Simplifying modulo 8, we get $(7x + 6 \leq x + z) \wedge \exists y.\, ((2y = 5x + 2) \wedge (y \neq 6x + 7z) \wedge (5y + z \leq 5x + 2))$. Note that the result is of the form $C_1 \wedge \exists x.\, C_2$, as specified in Lemma 1.

Our QE algorithm for conjunctions of LMCs uses a layered approach. Relatively less complete but sound and cheap heuristics are invoked first, and more complete but expensive techniques are used only when required. We now outline heuristic *QE1_Layer1* that forms the crux of the first (and also the cheapest) layer. Given a conjunction of LMCs A and a variable x to be quantified, *QE1_Layer1* computes $\exists x.\, A$ as $C_1 \wedge \exists x.\, C_2$ based on Lemma 1. If the $\kappa(x, \cdot)$ of all LMDs and LMIs in A are at least as large as k_1 (as in Lemma 1), then C_2 consists of the single LME $2^{k_1} \cdot x = t_1$. In this case, $\exists x.\, C_2$ simplifies to $2^{p-k_1} \cdot t_1 = 0$, and *QE1_Layer1* suffices to compute $\exists x.\, A$. However, in general,

C_2 may contain LMDs and LMIs with $\kappa(x, \cdot)$ values less than k_1. We describe techniques to address such cases in the following subsections.

2.1 Identifying Unconstraining LMIs and LMDs

Our goal in this subsection is to express C_2, obtained after application of *QE1_Layer1*, as $C \wedge D \wedge I$, where (i) D is a conjunction of (zero or more) LMDs in C_2, (ii) I is a conjunction of (zero or more) LMIs in C_2, (iii) C is the conjunction of the remaining LMCs in C_2, and (iv) $\exists x. (C) \Rightarrow \exists x. (C \wedge D \wedge I)$. Since $\exists x. (C \wedge D \wedge I) \Rightarrow \exists x. (C)$ always holds, this would allow us to compute $\exists x. C_2$, or equivalently $\exists x. (C \wedge D \wedge I)$, as $\exists x. C$. We call D and I as "unconstraining" LMDs and LMIs, respectively, in such cases.

Given C, D and I satisfying conditions (i), (ii) and (iii) above, we first focus on finding sufficient and efficiently checkable conditions for condition (iv) to hold. Let $x[i]$ denote the i^{th} bit of a bit-vector x, where $x[0]$ denotes its least significant bit. For $i \leq j$, let $x[i : j]$ denote the slice of bit-vector x consisting of bits $x[i]$ through $x[j]$. Given slice $x[i : j]$, its value is the natural number encoded by the bits in the slice. A key notion used in the subsequent discussion is that of "engineering" a solution of a constraint to make it satisfy another constraint. Formally, we say that a solution θ_1 of a conjunction φ of LMCs can be engineered with respect to slice $x[i : j]$ to satisfy a (possibly different) conjunction ψ of LMCs if there exists a solution θ_2 of ψ that matches θ_1 except possibly in the bits of slice $x[i : j]$. The central idea in the second layer of our QE algorithm is to efficiently compute an under-approximation η of the number of ways in which an arbitrary solution of C can be engineered to satisfy $C \wedge D \wedge I$. It is easy to see that if $\eta \geq 1$, then $\exists x. (C) \Rightarrow \exists x. (C \wedge D \wedge I)$.

Let I be $\bigwedge_{i=1}^{n}(l_i)$, where each l_i is an LMI of the form $s_i \bowtie t_i$, the operator $\bowtie$ is in $\{\leq, \geq\}$, s_i is a linear term with x in its support, and t_i is a linear term free of x. Note that this implies some loss of generality, since we disallow LMIs of the form $s \bowtie t$, where both s and t have x in their support. However, our experiments indicate that this is not very restrictive in practice. Let $s_1, \ldots, s_r$ be the distinct linear terms in I with x in their support. We partition I into $I_1, \ldots, I_r$, where each I_j is the conjunction of only those LMIs in I that contain the linear term s_j. We assume without loss of generality that each I_j contains the trivial LMIs $s_j \geq 0$ and $s_j \leq 2^p - 1$. Let I_j have n_j LMIs, of which the first $m_j (< n_j)$ are of the form $s_j \geq t_q$, where $1 \leq q \leq m_j$. Let the remaining LMIs in I_j be of the form $s_j \leq t_q$, where $m_j + 1 \leq q \leq n_j$.

Consider the inequality $Z_j : u_j \leq s_j \leq v_j$, where u_j denotes $\max_{q=1}^{m_j}(t_q)$ and v_j denotes $\min_{q=m_j+1}^{n_j}(t_q)$. Although Z_j is not a LMI, it is semantically equivalent to I_j. For notational convenience, let us denote $\kappa(x, s_j)$ by k_j. Clearly, the value of slice $x[p - k_j : p - 1]$ does not affect the satisfaction of Z_j. We wish to compute the number of ways, say N_j, in which an arbitrary solution of C can be engineered with respect to slice $x[0 : p - k_j - 1]$ to satisfy Z_j. Towards this end, we compute an integer δ_j in $\{0, \ldots, 2^p - 1\}$ such that $\delta_j \leq v_j - u_j + 1$. Intuitively, δ_j represents the minimum number of *consecutive* values that s_j can take for every combination of values of other variables, if we were to treat s_j as a fresh p-bit variable and if Z_j were to be satisfied. In general, however, s_j is of the form $a_j \cdot x + w_j$, where w_j is a linear term free of x, and a_j is a multiple of 2^{k_j}. Therefore, for every combination of values of variables other than x, there exist at least $\lfloor \delta_j / 2^{k_j} \rfloor$ consecutive values that $x[0 : p - k_j - 1]$ can take while satisfying Z_j. Hence, $N_j \geq \lfloor \delta_j / 2^{k_j} \rfloor$. For notational convenience, let us denote $\lfloor \delta_j / 2^{k_j} \rfloor$ by $\widehat{N_j}$.

To understand how δ_j is computed, recall that for every g in $\{1 \ldots m_j\}$ and for every h in $\{m_j + 1 \ldots n_j\}$, we have $t_g \leq s_j \leq t_h$. For every such pair of indices g and h, let $\delta_{g,h}$ be an integer in $\{0, \ldots, 2^p - 1\}$ such that $\delta_{g,h} \leq t_h - t_g + 1$. The value of δ_j can then be obtained as the minimum of all $\delta_{g,h}$ values. For reasons of simplicity and efficiency, we compute the values of $\delta_{g,h}$ conservatively as follows: (i) if t_g and t_h are constants, then $\delta_{g,h} = \max(t_h - t_g + 1, 0)$, (ii) if t_h is a constant and t_g can be expressed as $2^\tau \cdot t$, where $\tau \in \{0, 1, \ldots, p-1\}$, then $\delta_{g,h} = \max(t_h - (2^p - 2^\tau) + 1, 0)$, (iii) if t_g is a constant and t_h can be expressed as $2^\tau \cdot t + a$, where $\tau \in \{0, 1, \ldots, p-1\}$, then $\delta_{g,h} = \max(a \bmod 2^\tau - t_g + 1, 0)$, and (iv) $\delta_{g,h} = 0$ otherwise.

Let D be $\bigwedge_{i=1}^{m}(d_i)$, where each d_i is an LMD. Let k_0 denote $\kappa(x, C)$, and let C be such that k_0 is greater than both $\max_{i=1}^{m} \kappa(x, d_i)$ and $\max_{j=1}^{r} k_j$ (recall that $k_j = \kappa(x, s_j)$). To simplify the exposition, suppose further that $k_1 > \ldots > k_r$. We partition the bits of x into $r + 2$ slices as shown in Fig. 1, where slice$_0$ represents $x[0 : p - k_0 - 1]$, slice$_j$ represents $x[p - k_{j-1} : p - k_j - 1]$ for $1 \leq j \leq r$, and slice$_{r+1}$ represents $x[p - k_r : p - 1]$. Note that the value of slice$_0$ potentially affects the satisfaction of C as well as that of Z_1 through Z_r, the value of slice$_j$ potentially affects the satisfaction of Z_j through Z_r for $1 \leq j \leq r$, and the value of slice$_{r+1}$ does not affect the satisfaction of any Z_j or C.

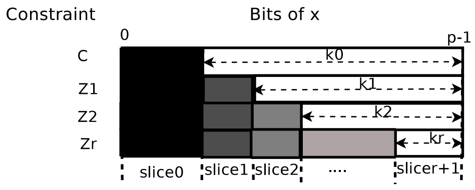


Fig. 1. Slicing of bits of x by $k_0, \ldots, k_r$

We have already seen that for every combination of values of variables other than x, there exist at least $\widehat{N_j}$ consecutive values that can be assigned to $x[0 : p - k_j - 1]$, while satisfying Z_j. Thus, if Z_0 denotes True, and if θ is a solution of $C \wedge \bigwedge_{i=0}^{j-1} Z_i$, where $0 \leq i < j \leq r$, then there exist at least $\lfloor \widehat{N_j} / 2^{p-k_i} \rfloor$ consecutive values that can be assigned to the slice $x[p - k_i : p - k_j - 1]$ while satisfying Z_j. Since slice$_0$ through slice$_i$ are unchanged, each such engineered solution must also satisfy $C \wedge \bigwedge_{i=0}^{j-1} Z_i$.

Let $Y_{i,j}$ denote the number of ways in which an arbitrary solution of $C \wedge \bigwedge_{i=0}^{j-1} Z_i$ can be engineered with respect to bits in slice$_{i+1}$ through slice$_j$, to satisfy $C \wedge \bigwedge_{i=0}^{j} Z_i$. By the argument given above, $Y_{i,j} \geq \lfloor \widehat{N_j} / 2^{p-k_i} \rfloor$, and the values of $x[p - k_i : p - k_j - 1]$ in the corresponding engineered solutions are consecutive. The latter fact implies that if we focus only on slice$_{i+1}$, then there are at least $\min(\lfloor \widehat{N_j} / 2^{p-k_i} \rfloor, 2^{k_i - k_{i+1}})$ consecutive values of slice$_{i+1}$ in the corresponding engineered solutions. Note that the min expression is necessary since

slice_{i+1} can only have one of $2^{k_i-k_{i+1}}$ distinct values. For notational convenience, let us denote $\min(\lfloor\widehat{N_j}/2^{p-k_i}\rfloor, 2^{k_i-k_{i+1}})$ by $\alpha_{i,j}$.

The above argument indicates that a solution θ of $C \wedge \bigwedge_{i=0}^{j-1} Z_i$ can be engineered to satisfy $C \wedge \bigwedge_{i=0}^{j} Z_i$ by using at least $\alpha_{i,j}$ different consecutive values of slice_{i+1}, for $0 \leq i < j \leq r$. Let the corresponding set of values of slice_{i+1} be denoted $S^{\theta}_{i+1,j}$. If $\bigcap_{j=i+1}^{r} S^{\theta}_{i+1,j}$ is non-empty, there exists a common value of slice_{i+1} that permits us to engineer θ with respect to slice_{i+1} through slice_r to satisfy Z_{i+1} through Z_r, respectively. It is therefore desirable to have $|\bigcap_{j=i+1}^{r} S^{\theta}_{i+1,j}| \geq 1$. Using the Inclusion-Exclusion principle, we find that $|\bigcap_{j=i+1}^{r} S^{\theta}_{i,j}| \geq (\sum_{j=i+1}^{r} \alpha_{i,j}) - (r-i-1)\cdot 2^{k_i-k_{i+1}}$. Note that the lower bound is independent of θ. For notational convenience, let us denote the lower bound by W_i.

If $W_i \geq 1$ for all $i \in \{1,\ldots r\}$, an arbitrary solution θ of C can be engineered to satisfy $C \wedge \bigwedge_{i=1}^{r} Z_i$ as follows. Since $W_1 \geq 1$, we choose a value of slice_1, say v_1, from $\bigcap_{j=1}^{r} S^{\theta}_{1,j}$. Let θ_1 denote θ with slice_1 (possibly) changed to have value v_1. Then θ_1 satisfies $C \wedge Z_1$. Since $W_2 \geq 1$, we can now choose a value of slice_2, say v_2, from $\bigcap_{j=2}^{r} S^{\theta_1}_{2,j}$, and repeat the procedure until we have chosen values for slice_1 through slice_r. Finally, since slice_{r+1} does not affect the satisfaction of C or of any Z_i, we can choose an arbitrary value for slice_{r+1}. Clearly, there are at least $(\prod_{i=0}^{r-1}|W_i|)\cdot 2^{k_r}$ ways in which values of different slices can be chosen, so as to engineer θ to satisfy $C \wedge \bigwedge_{i=1}^{r} Z_i$. Let us denote $(\prod_{i=0}^{r-1}|W_i|)\cdot 2^{k_r}$ by μ_I.

For every combination of values of variables other than x, let μ_D be an over-approximation of the number of values that can be assigned to slice_0 through slice_{r+1} such that D is violated. As shown in [1], $\mu_D = \sum_{i=1}^{m}(2^{\kappa(x,d_i)})$. Thus, we have at least $\mu_I - \mu_D$ ways of assigning values to slice_1 through slice_{r+1} when engineering a solution of C to satisfy $C \wedge D \wedge \bigwedge_{i=1}^{r} Z_i$. The details of extending these ideas to the general case, where $k_1 \geq \ldots \geq k_r$ can be found in [14].

Lemma 2. *If* $\eta = \mu_I - \mu_D \geq 1$*, then* $\exists x.\,(C \wedge D \wedge I) \equiv \exists x.\,(C)$

Example: Consider the problem of computing $\exists x.\,((z = 4x+y) \wedge (6x+y \leq 4) \wedge (x \neq z))$ with modulus 8. Suppose $C \equiv (z = 4x+y)$, $D \equiv (x \neq z)$, and $I \equiv (6x+y \leq 4)$. Here $p = 3$, $k_0 = 2$, $k_1 = 1$, $r = 1$, $\delta_1 = 5$, and $\mu_D = 1$. Therefore $W_0 = \alpha_{0,1} = Y_{0,1} = 1$, and $\mu_Z = |W_0|\cdot 2^1 = 2$. Hence $\eta = 1$, which implies that $\exists x.\,(C \wedge D \wedge I) \equiv \exists x.\,(C)$.

We now present procedure *QE1_Layer2*, that applies the technique described above to problem instances of the form $\exists x.\,C_2$, obtained after invoking *QE1_Layer1*. *QE1_Layer2* initially expresses $\exists x.\,C_2$ as $\exists x.\,(C \wedge D \wedge I)$, where C denotes $2^{k_1}\cdot x = t_1$ and $D \wedge I$ denotes the conjunction of LMDs and LMIs in C_2. If η (as in Lemma 2) is at least 1, then $D \wedge I$ is dropped from C_2. Otherwise, the LMCs in $D \wedge I$ with the largest $\kappa(x,\cdot)$ value (i.e. LMCs whose satisfaction depends on the least number of bits of x) are identified and included in C, and the above process repeats. If all the LMIs and LMDs in $\exists x.\,C_2$ are dropped in this manner, then $\exists x.\,C_2$ reduces to $\exists x.\,(2^{k_1}\cdot x = t_1)$, and *QE1_Layer2* can return the equivalent form $2^{p-k_1}\cdot t_1 = 0$. Otherwise, *QE1_Layer2* returns $\exists x.\,C_3$, where C_3 is a conjunction of possibly fewer LMCs compared to C_2, such that

$\exists x. C_3 \equiv \exists x. C_2$. The next subsection describes techniques to eliminate quantifiers from such problem instances.

2.2 Fourier-Motzkin Elimination for LMIs

In this subsection, we present a Fourier-Motzkin (FM) style QE technique for conjunctions of LMIs. There are two obvious problems when trying to apply FM elimination for reals [3] to a conjunction of LMIs. Recall that FM elimination "normalizes" each inequality l w.r.t. the variable x being quantified by expressing l in an equivalent form $x \bowtie t$, where $\bowtie \in \{\leq, \geq\}$ and t is a term free of x. However, normalizing an LMI w.r.t. a variable requires greater care, since standard equivalences used for normalizing inequalities over reals do not carry over to LMIs [2]. Moreover, due to the lack of density of integers, FM elimination cannot be directly lifted to normalized LMIs. This motivates us to (i) define a weak normal form for LMIs, and (ii) adapt FM elimination to achieve QE from normalized LMIs.

Note that Omega Test [8] also defines a normal form for inequalities over integers, and adapts FM elimination over reals for QE from normalized inequalities over integers. However, our experiments indicate that our approach convincingly outperforms Omega Test.

A Weak Normal Form for LMIs: We say that an LMI l with x in its support is *normalized w.r.t.* x if it is of the form $a \cdot x \bowtie t$, or of the form $a \cdot x \bowtie b \cdot x$, where $\bowtie \in \{\leq, \geq\}$, and t is a linear term free of x. We will henceforth use $NF1$ to refer to the first normal form $(a \cdot x \bowtie t)$ and $NF2$ to refer to the second normal form $(a \cdot x \bowtie b \cdot x)$. A Boolean combination of LMCs φ is said to be normalized w.r.t. x if every LMI in φ with x in its support is normalized w.r.t. x.

We will now show that every LMI with x in its support can be equivalently expressed as a Boolean combination of LMCs normalized w.r.t. x. Before going into the details of normalizing LMIs, it would be useful to introduce some notation. We define $\Omega(t_1, t_2)$ as the condition under which $t_1 + t_2$ overflows a p-bit representation, i.e., $t_1 + t_2$ interpreted as an integer exceeds $2^p - 1$. Note that $\Omega(t_1, t_2)$ is equivalent to both $(t_2 \neq 0) \wedge (t_1 \geq -t_2)$ and $(t_1 \neq 0) \wedge (t_2 \geq -t_1)$.

Suppose we wish to normalize $x + 2 \leq y$ modulo 8 w.r.t. x. Noting that 6 is the additive inverse of 2 modulo 8, if $\Omega(x + 2, 6) \equiv \Omega(y, 6)$, then $(x + 2 \leq y) \equiv (x \leq y + 6)$ holds; otherwise $(x + 2 \leq y) \equiv (x > y + 6)$ holds. Note that $\Omega(x+2, 6) \equiv \Omega(y, 6)$ can be equivalently expressed as $(x \leq 5) \equiv (y \geq 2)$. Hence, $(x + 2 \leq y)$ can be equivalently expressed in the normalized form $\mathsf{ite}(\varphi, (x \leq y + 6), (x > y + 6))$, where φ denotes $(x \leq 5) \equiv (y \geq 2)$, and $\mathsf{ite}(\alpha, \beta, \gamma)$ is a shorthand for $(\alpha \wedge \beta) \vee (\neg\alpha \wedge \gamma)$. The Ω predicate thus allows us to perform a case-split and normalize each branch. The following Lemma generalizes this idea.

Lemma 3. *Let* $l_1 : (a \cdot x + t_1 \leq b \cdot x + t_2)$ *be an LMI, where* t_1 *and* t_2 *are linear terms without* x *in their supports. Then,* $l_1 \equiv \mathsf{ite}(\varphi, l_2, \neg l_2)$*, where* $l_2 \equiv (a \cdot x - b \cdot x \leq t_2 - t_1)$*, and* φ *is a Boolean combination of LMCs normalized w.r.t.* x*.*

Modified FM for Normalized LMIs: We begin by illustrating the primary idea through an example. Consider the problem of computing $\exists x.\, C$, where $C \equiv (y \le 4x) \wedge (4x \le z)$ with modulus 16. Note that $\exists x.\, C$ is "the condition under which there exists a multiple of 4 between y and z, where $y \le z$". It can be shown that $\exists x.\, C$ is true iff one of the following three conditions holds: (i) $(y \le z)$, and y is a multiple of 4, i.e., $(y \le z) \wedge (4y = 0)$, (ii) $(y \le z) \wedge (y \le 12) \wedge (z \ge y+3)$, (iii) $(y \le z)$, $(z < y+3)$, and $(y > z \pmod 4)$), i.e., $(y \le z) \wedge (z < y+3) \wedge (4y > 4z)$. Hence $\exists x.\, C$ is equivalent to $(y \le z) \wedge \varphi$, where φ is the disjunction of the following three formulas: (i) $(4y = 0)$, (ii) $(z \ge y+3) \wedge (y \le 12)$, (iii) $(z < y+3) \wedge (4y > 4z)$. Note that if x, y, z were reals, we would have obtained $(y \le z)$ for $\exists x.\, C$. However, this would over-approximate $\exists x.\, C$ in the case of fixed width bit-vectors. The following Lemma generalizes this idea.

Lemma 4. *Let $l_1 : (t_1 \le a \cdot x)$ and $l_2 : (a \cdot x \le t_2)$ be LMIs in NF1 w.r.t. x. Let k be $\kappa(x, a \cdot x)$. Then, $\exists x.\, (l_1 \wedge l_2) \equiv (t_1 \le t_2) \wedge \varphi$, where φ is the disjunction of the formulas: (i) $(2^{p-k} \cdot t_1 = 0)$, (ii) $(t_2 \ge t_1 + 2^k - 1) \wedge (t_1 \le 2^p - 2^k)$, and (iii) $(t_2 < t_1 + 2^k - 1) \wedge (2^{p-k} \cdot t_1 > 2^{p-k} \cdot t_2)$.*

Suppose we wish to compute $\exists x.\, I$, where I is a conjunction of LMIs normalized w.r.t. x. Let $I \equiv I_1 \wedge I_2$, where I_1 is the conjunction of LMIs in I that are in *NF1*, and I_2 is the conjunction of LMIs in I that are in *NF2*. In addition, let $a_1, \ldots, a_n$ be the distinct non-zero coefficients of x in LMIs in I_1, and let $I_{1,i}$ denote the conjunction of LMIs in I_1 in which the coefficient of x is a_i. Finally, let $\Delta(t_1, t_2, k)$ denote the result of computing $\exists x.\, ((t_1 \le a \cdot x) \wedge (a \cdot x \le t_2))$ using Lemma 4, where k denotes $\kappa(x, a \cdot x)$. It is easy to see that Lemma 4 can be used to compute $\exists x.\, I_{1i}$, for every $i \in \{1, \ldots n\}$. Similar to FM elimination, we partition the LMIs $l_{i,j} : a_i \cdot x \bowtie t_j$ in I_{1i} into two sets $\Lambda_\le$ and $\Lambda_\ge$, where $\Lambda_\bowtie = \{l_{i,j} \mid l_{i,j}$ is of the form $a_i \cdot x \bowtie t_j\}$, for $\bowtie \in \{\le, \ge\}$. We assume without loss of generality that the trivial LMIs $a_i \cdot x \le 2^p - 1$ and $a_i \cdot x \ge 0$ are present in $\Lambda_\le$ and $\Lambda_\ge$ respectively. We can now compute $\exists x.\, I_{1i}$ as $\bigwedge_{(a_i \cdot x \le t_p) \in \Lambda_\le,\, (a_i \cdot x \ge t_q) \in \Lambda_\ge} \left(\Delta\left(t_q, t_p, \kappa\left(x, a_i \cdot x\right)\right)\right)$.

Each conjunction of LMIs such as I_{1i} above, where all LMIs are in *NF1* w.r.t. x, and have the same coefficient of x are said to be "unified" w.r.t. x. A Boolean combination of LMCs φ is said to be unified w.r.t. x if all LMIs in φ with x in their support are in *NF1* w.r.t. x and have the same coefficient of x. Unfortunately, unifying I w.r.t. x is inefficient in general. Hence we propose unifying I w.r.t. x only in the following cases, where unification can be done efficiently: (a) $I_2 \equiv \mathsf{true}$, $n = 2$ and $a_2 = -a_1$, or (b) $I_2 \equiv \mathsf{true}$, and every a_i is of the form $2^{k_i} \cdot e$, where e is an odd number in $\{1, \ldots, 2^p - 1\}$ independent of i. In case (a) above, I can be equivalently expressed as a Boolean combination of LMCs unified w.r.t. x, by replacing each occurrence of a_2 by $-a_1$ using the equivalence $(-t_1 \le -t_2) \equiv (t_1 = 0) \vee ((t_2 \ne 0) \wedge (t_1 \ge t_2))$. Case (b) deserves some additional explanation.

Consider the problem of computing $\exists x.\, I$, where $I \equiv (y \le 2x) \wedge (x \le z)$ with modulus 8. It can be shown that $x \le z$ can be equivalently expressed as the disjunction of (i) $\Omega(x, x) \wedge \Omega(z, z) \wedge (2x \le 2z)$, (ii) $\neg\Omega(x, x) \wedge \neg\Omega(z, z) \wedge (2x \le 2z)$, and (iii) $\neg\Omega(x, x) \wedge \Omega(z, z)$. Hence, $\exists x.\, I$ can be equivalently expressed as $\exists x.\, \varphi'$, where φ' is the disjunction of (i) $\Omega(x,x) \wedge \Omega(z,z) \wedge (2x \leq 2z) \wedge (y \leq 2x)$, (ii) $\neg\Omega(x,x) \wedge \neg\Omega(z,z) \wedge (2x \leq 2z) \wedge (y \leq 2x)$, and (iii) $\neg\Omega(x,x) \wedge \Omega(z,z) \wedge (y \leq 2x)$. Note that $\Omega(x,x)$ and $\Omega(z,z)$ can be equivalently expressed as $x \geq 4$ and $z \geq 4$ respectively. However, on closer inspection, it can be seen that occurrences of $x \geq 4$ in $\exists x.\, \varphi'$ arising from $\Omega(x,x)$ are unconstraining, and can therefore be dropped. Thus $\exists x.\, \varphi'$ can be equivalently expressed as $\exists x.\, \varphi$, where φ is the disjunction of $(2x \leq 2z) \wedge (y \leq 2x)$ and $(z \geq 4) \wedge (y \leq 2x)$. Note that $\exists x.\, \varphi$ is equivalent to $\exists x.\, I$ and is unified w.r.t. x. In general, given $\exists x.\, I$ such that $I_2 \equiv \mathsf{true}$ and the a_i's have the same e (as defined above), we make use of the above idea for unifying I w.r.t. x such that $\max_{i=1}^{n}(a_i)$ is the coefficient of x in all LMIs involving x. More details can be found in [14]. Note that normalizing and unifying a given conjunction of LMIs w.r.t. a variable converts it to a Boolean combination of LMCs in general. We make use of one of the techniques in section 3 for eliminating quantifiers from such Boolean combinations of LMCs.

In cases other than those covered in (a) and (b) above, we propose computing $\exists x.\, I$ using *model enumeration*, i.e., by expressing $\exists x.\, I$ in the equivalent form $I|_{x \leftarrow 0} \vee \ldots \vee I|_{x \leftarrow 2^p - 1}$ where $I|_{x \leftarrow i}$ denotes I with x replaced by the constant i.

The procedure that computes $\exists x.\, C_3$ (where C_3 is obtained from *QE1_Layer2*) using techniques mentioned in this subsection is called *QE1_Layer3*. Initially, LMEs and LMDs in C_3 are converted to LMIs using the equivalences $(t_1 = t_2) \equiv (t_1 \geq t_2) \wedge (t_1 \leq t_2)$ and $(t_1 \neq t_2) \equiv \neg(t_1 = t_2)$. Subsequently, $\exists x.\, C_3$ is computed either by normalizing and unifying C_3 w.r.t. x, followed by QE from the resulting Boolean combination of LMCs, or by model enumeration.

Recall that *QE1_Layer1*, *QE1_Layer2*, and *QE1_Layer3* try to eliminate a single quantifier from a conjunction of LMCs. These can be easily extended to eliminate multiple quantifiers by invoking them iteratively. Thus we have procedures *Layer1*, *Layer2*, and *Layer3* - extensions of *QE1_Layer1*, *QE1_Layer2*, and *QE1_Layer3* respectively, to eliminate multiple quantifiers.

Finally, we present our overall QE algorithm *Project* for computing $\exists X.\, A$, where A is a conjunction of LMCs over a set of variables V such that $X \subseteq V$. Initially *Project* tries to compute $\exists X.\, A$ using *Layer1*. This reduces $\exists X.\, A$ to an equivalent conjunction of A_1 and $\exists Y.\, A_2$, where A_1, A_2 are conjunctions of LMCs and $Y \subseteq X$. If all variables in X are eliminated by *Layer1*, then $\exists X.\, A \equiv A_1$. *Project* returns A_1 in this case. Otherwise, *Project* tries to compute $\exists Y.\, A_2$ using *Layer2*. *Layer2* reduces $\exists Y.\, A_2$ to an equivalent conjunction of A_3 and $\exists Z.\, A_4$, where A_3, A_4 are conjunctions of LMCs and $Z \subseteq Y$. If all variables in Y are eliminated by *Layer2*, then $\exists X.\, A \equiv A_1 \wedge A_3$. *Project* returns $A_1 \wedge A_3$ in this case. Otherwise, *Project* calls *Layer3* to compute $\exists Z.\, A_4$, and returns $A_1 \wedge A_3 \wedge \exists Z.\, A_4$. *Layer3*

3 QE from Boolean Combinations of LMCs

In [1], we explored a *Decision Diagram (DD)*-based approach and an *SMT solving (SMT)*-based approach for extending a QE algorithm for conjunctions of

LMEs and LMDs to Boolean combinations of LMEs and LMDs. In this section, we extend these approaches to Boolean combinations of LMEs, LMDs and LMIs. We also present a hybrid approach that tries to combine the strengths of the DD-based and SMT-based approaches.

Extending DD-Based and SMT-Based Approaches: Linear Modular Decision Diagrams (LMDDs) [1] are BDD-like data structures used to represent Boolean combinations of LMEs and LMDs. By allowing nodes in LMDDs to be labeled with LMEs or LMIs, we can use LMDDs to represent Boolean combinations of LMEs, LMIs and LMDs. In the subsequent discussion, we represent a non-terminal LMDD node f as $(\mathit{pred}(f), \mathit{high}(f), \mathit{low}(f))$, where $\mathit{pred}(f)$ is the LME/LMI labeling the node, $\mathit{high}(f)$ is the high child, and $\mathit{low}(f)$ is the low child, as defined in [13]. For simplicity of notation, we will use f to denote both an LMDD and the Boolean combination of LMCs represented by it, when the context precludes any disambiguity in interpretation.

Given an LMDD f and a variable x, the DD-based approach for computing $\exists x. f$ is similar to that described in [1]. Specifically, we perform a recursive traversal of the LMDD f, collecting the set of LMCs containing x (henceforth called *context*) encountered along the path from the root node of LMDD f. We call the corresponding recursive procedure *QE1_LMDD*. In each recursive call, *QE1_LMDD* computes an LMDD for $\exists x. (g \wedge C_x)$, where g is the LMDD encountered during the traversal and C_x is the conjunction of LMCs in the context. If g is a 1-terminal, then $\exists x. (g \wedge C_x)$ is computed by calling *Project* on $\exists x. C_x$. If the root node of g is a non-terminal, then *QE1_LMDD* simplifies g using the LMEs in C_x before traversing g, as described in[1]. Multiple variables can be eliminated by invoking *QE1_LMDD* repeatedly; this is implemented in the procedure *QE_LMDD*. The reader is referred to [14] for additional details of *QE_LMDD*.

In [1], we also proposed a procedure called *Monniaux* (originally introduced in [9]) that uses SMT solving to eliminate quantifiers from Boolean combinations of LMEs and LMDs. We extend *Monniaux* to handle Boolean combinations of LMCs involving LMIs. Suppose we wish to compute $\exists X. f$, where f is a Boolean combination of LMCs over a set of variables V and $X \subseteq V$. A naive way of computing this is by converting f to DNF by enumerating all satisfying assignments, and by invoking *Project* on each conjunction of LMCs. *Monniaux* improves upon this by generalizing a satisfying assignment to obtain a cube of satisfying assignments, by projecting the cube on the remaining variables (not in X), and then conjoining its complement with f before additional satisfying assignments are found.

Combining DD-Based and SMT-Based Approaches: The factors that contribute to the success of the DD-based approach are the presence of large shared sub-LMDDs and the strategy of eliminating one variable at a time. Both factors contribute to significant opportunities for reuse of results through dynamic programming. The success of the SMT-based approach is attributable primarily to pruning of the solution space achieved by interleaving of projection and model enumeration. In the following discussion, we present a hybrid approach that tries to combine the strengths of these two approaches.

The hybrid procedure, called *QE_Combined*, is shown in Fig. 2. The procedure uses the following helper functions: a) *qeddContext*: variant of *QE_LMDD* to compute $\exists X.\,(f \wedge C)$, where f is an LMDD and C is a conjunction of LMCs over a set of variables V, and $X \subseteq V$, b) *getConjunct*: computes the conjunction of LMCs in a given set, c) *Sat*: checks if a given Boolean combination of LMCs is satisfiable.

```
QE_Combined(f, X)
$\pi \leftarrow selectPath(f)$;
$S \leftarrow \emptyset$; /* set of sub-problems */
$simplify(f, \pi, S, \emptyset)$;
$g \leftarrow$ false;
for each ($\langle f_i, C_i\rangle \in S$)
  if ($Sat(f_i \wedge C_i \wedge \neg g)$)
    $h \leftarrow qeddContext(f_i, C_i, X)$;
    $g \leftarrow g \vee h$;
return $g$;
```

```
simplify(f, π, S, C)
/* $C$ : set of LMCs encountered along $\pi$ */
if (node $f$ is a terminal)
  $S \leftarrow S \cup \{\langle f, getConjunct(C)\rangle\}$;
else if (node $high(f)$ is in $\pi$)
  $S \leftarrow S \cup \{\langle low(f), getConjunct(C)\wedge\neg pred(f)\rangle\}$;
  $simplify(high(f), \pi, S, C \cup \{pred(f)\})$;
else /* node $low(f)$ is in $\pi$ */
  $S \leftarrow S \cup \{\langle high(f), getConjunct(C)\wedge pred(f)\rangle\}$;
  $simplify(low(f), \pi, S, C \cup \{\neg pred(f)\})$;
```

Fig. 2. Algorithms *QE_Combined* and *simplify*

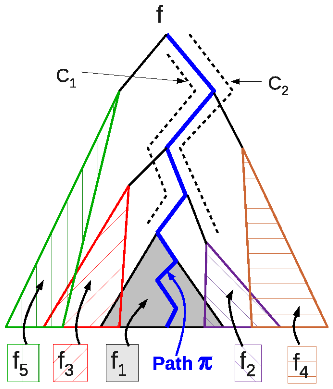


Fig. 3. Deriving $f_i \wedge C_i$ from path π

Procedure *QE_Combined* first selects a satisfiable path π in the LMDD f using a function *selectPath*. Subsequently, the procedure *simplify* is invoked, which traverses the path π, in order to convert (split) f into an equivalent disjunction $\bigvee_{i=1}^{n}(f_i \wedge C_i)$, where f_i denotes an LMDD and C_i denotes a conjunction of LMCs (represented in Fig. 2 as a set S of pairs, where each pair is of the form $\langle f_i, C_i\rangle$). Fig. 3(b) illustrates the splitting scheme followed by *simplify*. *QE_Combined* now computes $g \equiv \exists X.\,f$ as $\bigvee_{i=1}^{n}(\exists X.\,(f_i \wedge C_i))$ in the following manner: if $f_i \wedge C_i \wedge \neg g$ is satisfiable, then $h \equiv \exists X.\,(f_i \wedge C_i)$ is computed using *qeddContext*, and then h is disjoined with g. Note that unlike *Monniaux*, *QE_Combined* does not explicitly interleave projections inside model enumeration. However disjoining the result of $\exists X.\,(f_i \wedge C_i)$ with g, and computing $\exists X.\,(f_i \wedge C_i)$ only if $f_i \wedge C_i \wedge \neg g$ is satisfiable helps in pruning the solution space of the problem, as achieved in *Monniaux*.

4 Experimental Results

We performed experiments to (i) evaluate the performance of *Monniaux*, *QE_LMDD*, and *QE_Combined*, (ii) evaluate the effectiveness of the layers in *Project*, and (iii) compare the performance of *Project* with alternative QE techniques. The experiments were performed on a 1.83 GHz Intel(R) Core 2 Duo machine with 2GB memory running Linux, with a timeout of 1800 seconds. We implemented our own LMDD package for carrying out QE experiments involving LMDDs.

Benchmarks: We used a benchmark suite consisting of 198 *lindd* benchmarks [4] and 23 *vhdl* benchmarks. Each of these benchmarks is a Boolean combination of LMCs with a subset of the variables in their support existentially quantified.

The *lindd* benchmarks reported in [4] are Boolean combinations of octagonal constraints over integers, i.e., constraints of the form $a \cdot x + b \cdot y \leq k$ where x, y are integer variables, k is an integer constant, and $a, b \in \{-1, 1\}$. We converted these benchmarks to Boolean combinations of LMCs by assuming the size of integer as 16 bits. Although these benchmarks had no LMEs explicitly, they contained LMEs encoded as conjunctions of the form $(x - y \leq k) \wedge \neg(x - y \leq k - 1)$. We converted each such conjunction to an LME $x - y = k$ as a pre-processing step. The total number of variables, the number of variables to be eliminated, and the number of bits to be eliminated in the *lindd* benchmarks ranged from 30 to 259, 23 to 207, and 368 to 3312 respectively.

The *vhdl* benchmarks were obtained in the following manner. We took a set of word-level VHDL designs. Some of these are publicly available designs obtained from [5], and the remaining are proprietary. We derived the symbolic transition relations of these VHDL designs. The *vhdl* benchmarks were obtained by quantifying out all the internal variables (i.e. neither input nor output of the top-level module) from these symbolic transition relations. Effectively this gives abstract transition relations of the designs. The coefficients of the variables in these benchmarks were largely odd. These benchmarks contained a significant number of LMEs (arising from assignment statements in the VHDL programs). The total number of variables, the number of variables to be eliminated, and the number of bits to be eliminated in the *vhdl* benchmarks ranged from 10 to 50, 2 to 21, and 10 to 672 respectively.

Evaluation of *Monniaux*, *QE_LMDD*, and *QE_Combined*: We measured the time taken by *Monniaux*, *QE_LMDD*, and *QE_Combined* for QE from each benchmark. For *QE_LMDD* and *QE_Combined*, this included the time to build the initial LMDD. We observed that each approach performed better than the others for some benchmarks (see Fig. 4). Note that the points in Fig. 4(a) are scattered, while the points in Fig. 4(b) and 4(c) are more clustered near the 45° line. This shows that *DD* and *SMT* based approaches are incomparable, whereas the hybrid approach inherits the strengths of both *DD* and *SMT* based approaches. Hence, given a problem instance, we recommend the hybrid approach, unless the approach which will perform better is known a-priori.

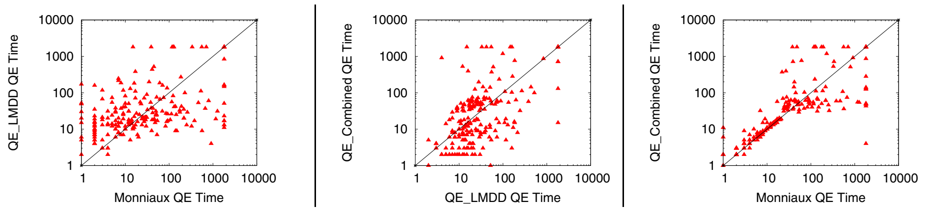


Fig. 4. Plots comparing (a) *Monniaux* and *QE_LMDD*, (b) *QE_LMDD* and *QE_Combined*, and (c) *Monniaux* and *QE_Combined* (All times are in seconds)

Evaluation of *Project*: Recall that *Layer3* converts a conjunction of LMCs to a Boolean combination of LMCs and calls *Monniaux*/ *QE_LMDD*/ *QE_Combined* for QE from this Boolean combination, which results in new (recursive) *Project* calls. Hence two kinds of *Project* calls were generated while performing QE from the benchmarks: (i) the initial/original *Project* calls, and the (ii) aforementioned recursive *Project* calls. In the subsequent discussion, whenever we mention "*Project* calls", it refers to the initial/original *Project* calls, unless stated otherwise.

Table 1. Details of *Project* calls (figures are per *Project* call)

Type	**Vars**	**Qnt**	**LMIs**	**LMEs**	**LMDs**	**Contr**			**Time**			
						L1	**L2**	**L3**	**L1**	**L2**	**L3**	***Pr***
lindd	39.9	38.1	(88, 0, 18.9)	(60, 0, 10.1)	(35, 0, 8.1)	51	44	5	3	5	13149	674
vhdl	9.3	7.8	(4, 0, 0.4)	(16, 0, 6.3)	(31, 0, 1.8)	95	4.5	0.5	1	6	161	3

Vars : Average number of variables, **Qnt** : Average number of quantifiers, **LMIs** : (Maximum, minimum, average) number of LMIs, **LMEs** : (Maximum, minimum, average) number of LMEs, **LMDs** : (Maximum, minimum, average) number of LMDs, **Contr** : Average contribution of a layer, **L1** : *Layer1*, **L2** : *Layer2*, **L3** : *Layer3*, **Pr** : *Project*, **Time** : Average time spent per quantifier eliminated in milli seconds

The total number of *Project* calls generated from the *lindd* and *vhdl* benchmarks were 52, 836 and 7, 335 respectively. Statistics of these *Project* calls are shown in Table 1. The contribution of a layer is measured as the ratio of the number of quantifiers eliminated by the layer to the number of quantifiers to be eliminated in the *Project* call, multiplied by 100. The contributions of the layers and the times taken by the layers per quantifier eliminated for individual *Project* calls from *lindd* benchmarks are shown in Fig. 5 and Fig. 6. The *Project* calls here are sorted in increasing order of contribution from *Layer1*.

Layer1 and *Layer2* were cheap and eliminated a large fraction of quantifiers in both *lindd* and *vhdl* benchmarks. This underlines the importance of our layered framework. The relatively large contribution of *Layer1* in the *Project* calls from *vhdl* benchmarks was due to significant number of LMEs in these problem instances. *Layer3* was found to be the most expensive layer. Most of the time spent in *Layer3* was consumed in the recursive *Project* calls. No *Layer3* call in our experiments required model enumeration. The large gap in the time per

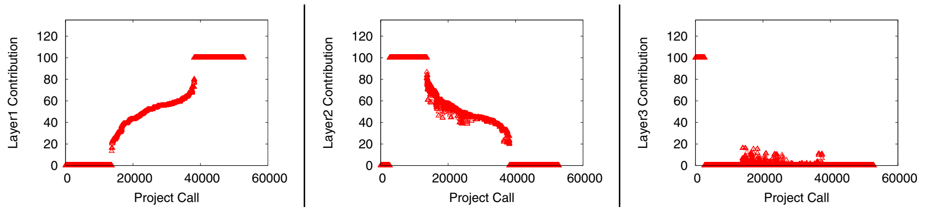


Fig. 5. Contribution of (a) *Layer1*, (b) *Layer2*, and (c) *Layer3* for *lindd* benchmarks

quantifier in *Layer2* and that in *Layer3* for both sets of benchmarks points to the need for developing additional cheap layers between *Layer2* and *Layer3* as part of future work.

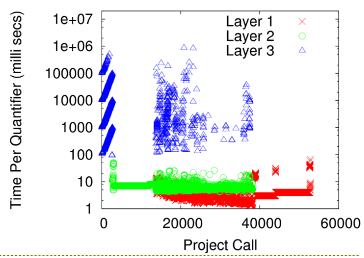


Fig. 6. Cost of layers for *lindd* benchmarks

Comparison of *Project* with alternative QE techniques: We compared the performance of *Project* with QE based on ILA using Omega Test, and also with QE based on bit-blasting. We implemented the following algorithms for this purpose: (i) *Layer1_Blast*: this procedure first quantifies out the variables using *Layer1* (recall that *Layer1* is a simple extension of [1]), and then uses bit-blasting and BDD based bit-level QE [6] for the remaining variables. (ii) *Layer1_OT*, *Layer2_OT*: *Layer1_OT* first quantifies out the variables using *Layer1*, and then uses conversion to ILA and Omega Test [8] for the remaining variables. *Layer2_OT* first quantifies out the variables using *Layer1* followed by *Layer2*, and then uses conversion to ILA and Omega Test for the remaining variables. *Layer2_OT* helps us to compare the performance of *Layer3* with that of Omega Test.

We collected the instances of QE problem for conjunctions of LMCs arising from *Monniaux* when QE is performed on each benchmark. We performed QE from such conjunction-level problem instances using *Project*, *Layer1_Blast*, *Layer1_OT*, and *Layer2_OT*. Fig. 7(a) and 7(b) compare the average QE times

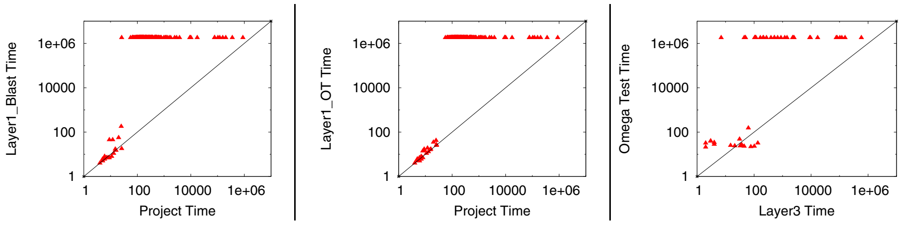


Fig. 7. Plots comparing (a) *Project* and *Layer1_Blast*, (b) *Project* and *Layer1_OT*, and (c) *Layer3* and Omega Test (All times are in milli seconds)

taken by *Project* against those taken by *Layer1_Blast* and *Layer1_OT* for QE from the conjunction-level problem instances for each benchmark. Subsequently, for each benchmark, we compared the average time consumed by *Layer3* in the *Project* calls with that consumed by Omega Test in the *Layer2_OT* calls (see Fig. 7(c)). The results clearly demonstrated that (i) *Project* outperforms both the alternative QE techniques and (ii) *Layer3* outperforms Omega Test. There were a few cases where Omega Test performed better than *Layer3*. This was due to the relatively larger number of recursive *Project* calls in these cases.

5 Conclusion

The need for efficient techniques for bit-precise quantifier elimination cannot be overemphasized. In this paper, we presented practically efficient techniques for eliminating quantifiers from Boolean combinations of LMCs. We propose to study quantifier elimination techniques for non-linear modular constraints as part of future work.

References

1. John, A.K., Chakraborty, S.: A Quantifier Elimination Algorithm for Linear Modular Equations and Disequations. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 486–503. Springer, Heidelberg (2011)
2. Bjørner, N., Blass, A., Gurevich, Y., Musuvathi, M.: Modular difference logic is hard. CoRR abs/0811.0987 (2008)
3. Kroening, D., Strichman, O.: Decision procedures: an algorithmic point of view. Texts in Theoretical Computer Science. Springer (2008)
4. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear arithmetic. In: FMCAD (2009)
5. ITC'99 benchmarks, `http://www.cad.polito.it/downloads/tools/itc99.html`
6. CUDD release 2.4.2 website, `vlsi.colorado.edu/~fabio/CUDD`
7. Enderton, H.: A mathematical introduction to logic. Academic Press (2001)
8. Pugh, W.: The Omega Test: A fast and practical integer programming algorithm for dependence analysis. Communications of the ACM, 102–114 (1992)
9. 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)
10. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
11. Jain, H., Clarke, E., Grumberg, O.: Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
12. Muller-Olm, M., Seidl, H.: Analysis of modular arithmetic. ACM Transactions on Programming Languages and Systems 29(5), 29 (2007)
13. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
14. John, A., Chakraborty, S.: Extending quantifier elimination to linear inequalities on bit-vectors, Technical Report TR-12-35, CFDVS, IIT Bombay, `http://www.cfdvs.iitb.ac.in/reports/reports/tacas2013_report.pdf`

The MathSAT5 SMT Solver [⋆]

Alessandro Cimatti[1], Alberto Griggio[1],
Bastiaan Joost Schaafsma[1,2], and Roberto Sebastiani[2]

[1] FBK-IRST, Trento, Italy
[2] DISI, University of Trento, Italy

Abstract. MATHSAT is a long-term project, which has been jointly carried on by FBK-IRST and University of Trento, with the aim of developing and maintaining a state-of-the-art SMT tool for formal verification (and other applications). MATHSAT5 is the latest version of the tool. It supports most of the SMT-LIB theories and their combinations, and provides many functionalities (like e.g. unsat cores, interpolation, AllSMT). MATHSAT5 improves its predecessor MATHSAT4 in many ways, also providing novel features: first, a much improved incrementality support, which is vital in SMT applications; second, a full support for the theories of arrays and floating point; third, sound SAT-style Boolean formula preprocessing for SMT formulae; finally, a framework allowing users for plugging their custom tuned SAT solvers. MATHSAT5 is freely available, and it is used in numerous internal projects, as well as by a number of industrial partners.

1 Introduction

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a (typically quantifier-free) first-order formula with respect to some decidable theory $\mathcal{T}$ (or combination of theories $\bigcup_i \mathcal{T}_i$). SMT solvers have proved to be powerful and expressive backend engines for formal verification in many contexts, including the verification of software, hardware, and of timed and hybrid systems. An amount of papers with novel and very efficient techniques for SMT has been published in the last decade, and some very efficient SMT tools are now available.

MATHSAT is a long-term project, which has been jointly carried on by FBK-IRST and University of Trento in the last decade, with the aim of developing and maintaining a state-of-the-art SMT tool for formal verification (and other applications). In this paper we present MATHSAT5, the latest version of the tool. MATHSAT5 supports most of the SMT-LIB theories and their combinations, and provides many SMT functionalities (e.g. unsatisfiable cores, interpolation, AllSMT). It does not offer support for quantifiers. MATHSAT5 improves its predecessor MATHSAT4 [5] in many ways, also providing novel features. First, it provides a much improved support for *incremental solving*, which is vital in many applications of SMT (e.g., symbolic simulation, SW model checking). Second, it fully supports also the theories of *arrays* and *IEEE floating*

[⋆] A. Griggio is 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. B. Schaafsma and R. Sebastiani are supported in part by Semiconductor Research Corporation under GRC Research Project 2012-TJ-2266 WOLF.

point numbers. Third, it provides (incremental) *Theory Aware SAT-Preprocessing*, i.e. sound SAT-style Boolean formula preprocessing adapted for SMT formulas. Finally, it supplies a framework for third-party SAT-solver integration, allowing users -including industrial users- for plugging their custom tuned solvers. MATHSAT5 is available at [29], and it is used in numerous internal projects, as well as by a number of industrial partners.

The paper is structured as follows: §2 describes the functionalities of MATHSAT5; §3 discusses its architecture; §4 discusses the specifics of our implementations; §5 shows an empirical evaluation; §6 discusses a number of in-house applications of MATHSAT5; finally in §7 we draw some conclusions and discuss ongoing and future work.

2 Functional View

MATHSAT5 provides functionalities for both satisfiability checking (*solving*) and for extended SMT tasks. It can be accessed either through the command line, by feeding a `SMT-LIB` file (in either the `SMT-LIB v.1` or `SMT-LIB v.2` standard), or through an API, which is similar in spirit to the commands of the `SMT-LIB v.2` language (with additional functionalities).

Solving. MATHSAT5 solving facilities support most of the `SMT-LIB` theories of interest, including that of equality and uninterpreted functions ($\mathcal{EUF}$), that of arrays ($\mathcal{AR}$), and their combinations with the theories of linear arithmetic on the rationals ($\mathcal{LA}(\mathbb{Q})$), the integers ($\mathcal{LA}(\mathbb{Z})$) and mixed rational-integer ($\mathcal{LA}(\mathbb{QZ})$), that of fixed-width bit-vectors ($\mathcal{BV}$), and that of floating-point arithmetic ($\mathcal{FP}$). Notably, to the best of our knowledge, MATHSAT5 is one of the very few SMT solvers supporting $\mathcal{FP}$.

Many SMT-based formal verification techniques (e.g., BMC, symbolic simulation, lazy abstraction) need invoking the backend SMT solver *incrementally*, in a stack-based manner, by pushing and popping sub-formulas. To cope with this fact, regardless the theories addressed, MATHSAT5 provides an *incremental*, stack-based interface, allowing multiple satisfiability checks over a changing clause database, and maintaining useful information of the status of computation (e.g. learned clause, scores) from one call to the other, which prevents restarting the search from scratch each time.

Beyond Solving. Like its predecessors, MATHSAT5 was designed primarily to be used in formal verification settings, where often simple queries for a "SAT/UNSAT" answer are not sufficient. Thus, MATHSAT5 provides several extended SMT functionalities.
Production of Models. When the input formula φ is satisfiable, MATHSAT5 can produce a satisfying interpretation $\mathcal{I}$ on domain variables, with a congruent partial interpretation of uninterpreted functions and predicates. [1]

Production of Proofs. When φ is unsatisfiable, MATHSAT5 can produce a proof, combining a resolution proof and theory-specific sub-proofs of the $\mathcal{T}$-lemmas.

Extraction of Unsatisfiable Cores. MATHSAT5 allows for extracting a $\mathcal{T}$-unsatisfiable subset of an input clause set. This implements both the standard extraction from a

[1] E.g., in $\mathcal{EUF} \cup \mathcal{LA}(\mathbb{Z})$, if φ is $x = 5 \wedge f(x) < 3$, then $\mathcal{I}$ may assign x to 5 and $f(5)$ to 2.

resolution proof, and the "lemma-lifting" approach we described in [10], which invokes an external Boolean unsat-core extractor available off-the-shelf, thus benefitting from every size-reduction techniques implemented there.

Interpolation. MATHSAT5 allows for computing (Craig) interpolants of pairs of input mutually-inconsistent SMT formulas for nearly all implemented theories. This feature includes optimized interpolant generator for $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Q})$ [9], for $\mathcal{LA}(\mathbb{Z})$ [27], for $\mathcal{BV}$ [26], and an interpolant generator for combined theories based on DTC [9].

AllSMT & Predicate Abstraction. MATHSAT5 implements an "AllSMT" functionality [28]: in case of a satisfiable input formula φ, it can efficiently enumerate a complete set of theory-consistent partial assignments satisfying φ. This feature is useful for performing predicate abstraction in a SMT-based Counter-Example-Guided Abstraction-Refinement (CEGAR) context (e.g. [3]).

Enumeration of Diverse Models. Strictly related to AllSMT, MATHSAT5 implements a brand-new functionality, which was requested from our industrial partners. The users are allowed to define a set of *diversifying predicates [resp. terms]* and MATHSAT5 enumerates models which differ to one another for the truth value [resp. domain value] of at least one of these predicates [resp. terms]. [2] This technique is useful to, e.g., guarantee coverage of all branches in a program, partitioning the value space into a grid, cover all values of some selector variables, investigate corner cases, etc.

Pluggable SAT Solvers. Finally, MATHSAT5 provides an API for integrating external SAT-solvers, allowing (industrial) users for plugging their custom tuned solver for their specific applications.

MathSAT5 vs. MathSAT4. MATHSAT5 extends and improves its predecessor MATHSAT4 in many ways.

From the perspective of SMT solving, a full support for the theories of arrays ($\mathcal{AR}$) and floating point ($\mathcal{FP}$) has been introduced; the solvers for $\mathcal{BV}$ and $\mathcal{LA}(\mathbb{Z})$ have been re-implemented and made much more efficient, and the latter has been extended to deal also with mixed rational-integers $\mathcal{LA}(\mathbb{QZ})$. The default underlying SAT solver has been improved. Moreover, (incremental) *Theory Aware SAT-Preprocessing*, i.e. sound SAT-style Boolean formula preprocessing adapted for SMT formulas, has been introduced. (See next sections.) Overall, the whole tool has been redesigned to fully support incrementality, in both solving and other functionalities.

From the perspective of SMT functionalities, *Enumeration of Diverse Models* and *Pluggable SAT Solvers* are brand new. *Interpolation* has been extended to $\mathcal{LA}(\mathbb{Z})$ [27] and $\mathcal{BV}$ [26]. Finally, the *Production of Models* and of *Production of Proofs* functionalities have been significantly improved. Importantly, the *Production of Proofs*, *Extraction of Unsat Cores*, *Interpolation*, *AllSMT*, *Enumeration of Diverse Models*, *Pluggable SAT Solvers* functionalities have been adapted to work also in incremental mode (*Production of Models* was already incremental in MATHSAT4).

[2] Notice that diversifying terms are meaningful only with terms on discrete and small bounded domains, like enumeratives, bounded integers with small ranges, small-size bit-vectors.

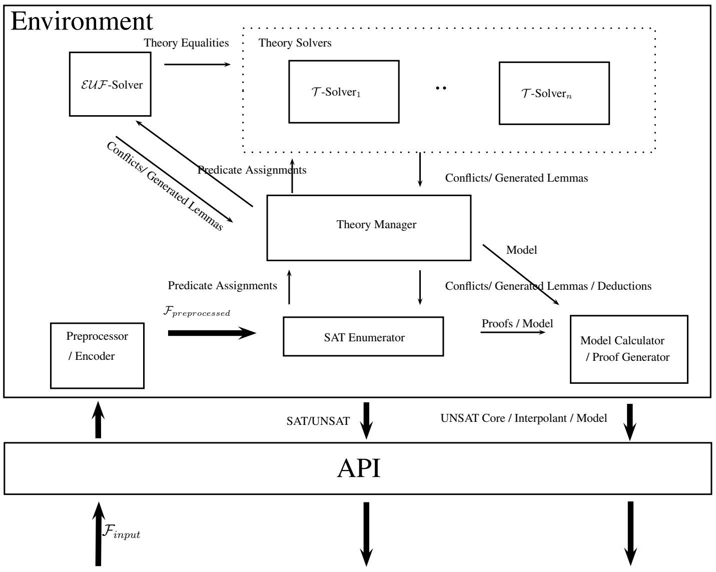


Fig. 1. Architectural overview of MATHSAT5

3 Architectural View

Figure 1 details the MATHSAT5 architecture. From a high-level perspective, the main component of MATHSAT5 is the environment, which acts as a coordinator for the various sub-components of the solver (preprocessor, constraint encoder, theory manager, proof and model generator, SAT engine and individual theory solvers). Besides coordination of components, the environment is also responsible for various administrative tasks, such as memory management and garbage collection.

The preprocessor is a term-rewriting engine which performs formula normalization and constant inlining. In formula normalization we rewrite redundant formulas to a "simpler" or "smaller" form. This is done by applying, up to a fix point, some rewrite rules from a database. In constant inlining we replace constants with their definitions. (For example if the formula contains a predicate $(x = 3)$, we replace all occurrences of x in the input formula by 3.)

The constraint encoder performs the CNF conversion of the input formulas, as well as the encoding of various constructs which are not directly supported by the core components of MATHSAT5. For instance, it eliminates term-level if-then-else constructs ITE(c, t, e) from a formula by replacing them with fresh variables x_{ITE} and by adding to the formula the clauses $(\neg c \vee (x_{\text{ITE}} = t))$ and $(c \vee (x_{\text{ITE}} = e))$.

The core of MATHSAT5 is composed of the SAT engine and the theory solvers, which interact following the standard lazy/DPLL($\mathcal{T}$) approach [2]. The SAT engine is either our native SAT engine or a "pluggable", third-party SAT engine. The former is a MINISAT-style SAT solver [19], equipped with a preprocessor/inprocessor supporting the following Boolean formula simplifications: *Variable Elimination (VE)*, *Subsumed* *clause removal (SCR)* and *Backwards subsumption (BS)* [18]. In VE we perform DP-resolution on a variable x, replacing all clauses of the form $(C \vee x)$ and $(C \vee \neg x)$ with their pairwise resolvents. In SCR, if clause C_i subsumes C_j, i.e. C_i contains a subset of the literals in C_j, then it follows the C_j can be dropped from the input formula. In BS, we take advantage from the fact that, if we resolve $(\neg x \vee C_i)$ with $(x \vee C_j)$ on x, and C_i subsumes C_j, then it follows that their resolvent equals C_j, thus we can shorten $(x \vee C_j)$ to C_j. Notice that in general (some of) these simplifications are unsound in an (incremental) SMT setting. We describe how we have adapted them to ensure correctness in §4.3.

The pluggable SAT engine allows for the integration of an external, third-party SAT solver in MATHSAT5. The architecture is based on a "SAT worker" wrapper interface for the external solver, which is required to implement a number of callback functions to respond to various events generated by the other MATHSAT5 components, and to satisfy certain requirements that are needed for a proper integration in an SMT context. For more details, we refer to §4.4.

The theory manager acts as a unified interface between the SAT engine and individual $\mathcal{T}$-solvers, allowing for a modular integration of new theories. In our architecture, individual $\mathcal{T}$-solvers know nothing about neither the SAT engine nor their sibling solvers, and they only interact with the theory manager. In this way, $\mathcal{T}$-solvers can be easily added and removed without affecting the rest of the system.

The SAT engine and the theory manager communicate with the model and proof calculator component, which is responsible of producing models for satisfiable formulas and refutation proofs for unsatisfiable ones. Refutation proofs consist of a Boolean part and a theory-specific part. The theory-specific part consists of the list of theory lemmas generated during search, together with theory-specific proofs for them. For example, for $\mathcal{LA}(\mathbb{Q})$ a proof consists of a list of inequalities and the corresponding coefficients needed for obtaining a contradiction via linear combinations, whereas for $\mathcal{EUF}$ it consists of a sequence of applications of the reflexivity, symmetry, transitivity and congruence axioms leading to the violation of some disequality. The Boolean part of the proof, computed by the SAT engine, consists instead of Boolean resolution steps among the clauses of (the CNF conversion of) the input formula and the theory lemmas generated by the $\mathcal{T}$-solvers. From the refutation proof, interpolants and/or unsatisfiable cores can then be produced (possibly with the help of an external Boolean unsat-core extractor, as described in [10]).

4 Implementation

In this Section, we provide some details on the most significant aspects of the implementation of MATHSAT5.

4.1 Low-Level Optimizations

MATHSAT5 is implemented in C++, using an object-oriented paradigm. One of the most important aspects of the implementation is the use of several ad-hoc variants of common data structures (such as vectors, stacks, queues, hash tables), specialized for critical parts of the code, which significantly improve the overall performance of the solver. The main reason for this is memory management. In particular, our custom data structures and algorithms are designed to reduce the overhead due to excessive memory allocations/deallocations and to exploit the availability of specialized allocators that try to ensure a cache-friendly layout of data in memory. For a similar reason, we use our own custom written library for arbitrary-precision arithmetic, built on top of the GNU Multi Precision library [23], which avoids costly memory operations in the cases in which the numbers to manipulate fit into machine words.

One might question the value of these low-level "micro-optimizations", arguing that there are many higher-level factors (such as e.g. branching heuristics, search strategies, preprocessing algorithms) which have a much stronger impact on the performance of an SMT solver. Our experience however suggests that in practice these details have a very visible impact, in particular on scalability, which is crucial for the successful application of the solver in industrial settings. We refer to [25] for an example of the impact of low-level optimizations on the performance of MATHSAT on real-world $\mathcal{LA}(\mathbb{Q})$ formulas.

4.2 Incrementality

In an incremental setting, MATHSAT5 manipulates a *stack* $S \overset{\text{def}}{=} [\varphi_1, \ldots, \varphi_n]$ of formulas, which corresponds to the input problem $\varphi_1 \wedge \ldots \wedge \varphi_n$. The stack is manipulated via a *push* and *pop* interface. Pushing a formula ψ corresponds to conjoining ψ to the current input problem, whereas popping corresponds to discarding the most recently added conjunct. All the internal components of MATHSAT outlined in Figure 1 are designed to exploit this stack-based interaction. In the DPLL engine, incrementality is implemented by exploiting a variant of solving under assumptions [20]. Each element φ_i of the stack is associated to a *label literal* x_{stack_i}. During CNF conversion, all the clauses for the formula φ_i are extended with the label literal $\neg x_{\text{stack}_i}$. When the satisfiability of the input formula is decided, DPLL is invoked with the assumptions $\{x_{\text{stack}_1}, \ldots, x_{\text{stack}_n}\}$. When a formula is popped from the stack, all clauses (including learnt clauses) that contain the last label literal $\neg x_{\text{stack}_n}$ are deleted. Importantly, all DPLL variables created after x_{stack_n} are also deleted, as well as all the corresponding internal variables in the theory solvers. This is very important in applications (such as e.g. [8]) in which hundreds of thousands of simple formulas, often totally unrelated to each other, are pushed and popped from the stack, in order to avoid cluttering the solver with irrelevant data.

4.3 Adapting SAT-Level Preprocessing to Incremental SMT

As stated above, MATHSAT5 supports the following SAT formula processing techniques: Variable Elimination (VE), Subsumed clause removal (SCR) and Backwards subsumption (BS). In general, these techniques are not sound when applied in an incremental SMT context. There are multiple reasons for this: 1) After model calculation, the extended SAT model which contains the values calculated for eliminated variables may be $\mathcal{T}$-inconsistent; 2) VE may eliminate label literals x_{stack_i} used for implementing incrementality; 3) Variables eliminated by VE may be reintroduced either during subsequent formula pushes, or during search; 4) Clauses which allowed us to shorten a clause through BS or eliminate a clause through SCR may no longer be implied by the input formula after a pop.

The first problem arises because, in an SMT context, variables in the SAT solver might represent theory constraints (i.e., they might be *proxies* for some $\mathcal{T}$-atom). In such cases, eliminating them has the effect of dropping some $\mathcal{T}$-constraints from the formula, which might change its satisfiability status. Our simple solution to this problem is to forbid the elimination of proxy variables (in SAT terminology, we *freeze* them).

The other three issues are not due to SMT, but rather to the use of the techniques in an incremental setting. Point 2) is problematic because label literals are necessary to correctly maintain the stack of formulas (see §4.2 above), and so they can't be eliminated from the formula. We avoid the problem by simply freezing label literals. For problem 3), we adopt a solution similar to the one described in [30]. Roughly speaking, the approach is based on saving clauses containing eliminated variables, instead of deleting them immediately, so that they can be re-added to the problem in case a previously-eliminated variable is re-added to the SAT solver. We simply remark that, unlike in the setting considered in [30], in SMT eliminated variables can be reintroduced even when incrementality is not used, because in general theory solvers are allowed to introduce new SAT variables during search (this is the case e.g. for Delayed Theory Combination [6] or for axiom instantiation [24]). Finally, regarding problem 4), we observe that freezing label literals automatically gives a solution for it. The reason is that, since we prohibit the elimination of label literals, clauses belonging to different pushes always differ in at least one literal which only occurs negatively, thus neither SCR nor BS is applicable. This solution, however, has the drawback of significantly limiting the applicability of subsumption. In fact, MATHSAT5 does something better than this, by employing the notions of contemporary and base clauses. Clause C_i is contemporary with respect to clause C_j if the highest label literal contained in C_i is created before the highest label literal contained in C_j. If C_i is contemporary to C_j, the push/pop architecture used in MATHSAT5 ensures that as long as C_j is active, C_i is active as well. Given a clause C_i, $base(C_i)$ is the clause obtained by removing all label literals from C_i. Using these notions of contemporary and base clauses MATHSAT5 extends the SCR and BS rules as follows:

– If $base(C_i)$ subsumes $base(C_j)$ and C_i is contemporary to C_j, we can drop C_j from the input formula.
– If $base(C_i)$ subsumes $base(C_j)$ but C_i is not contemporary to C_j, we can still ignore C_j as long as C_i is active.
– If $base(C_i)$ backwards subsumes $base(C_j)$ on l and C_i is contemporary to C_j, we can shorten C_j by l.

Figure 2 summarizes the clause management system used in MATHSAT5, and shows how clauses move from being active or locked, to inactive, or dropped, depending on the circumstances.

4.4 Pluggable SAT Solvers

As already described, MATHSAT5 allows for using an external CDCL-based SAT solver as its SAT engine. From the point of view of the implementation, this is achieved

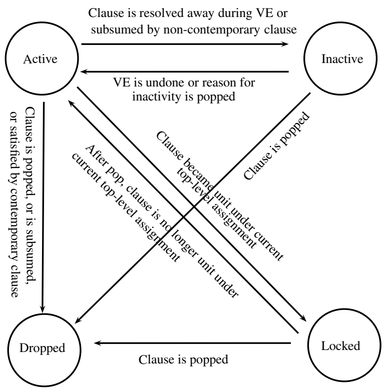

Active
Inactive
Dropped
Locked
Clause is resolved away during VE or subsumed by non-contemporary clause
VE is undone or reason for inactivity is popped
Clause is popped, or is subsumed, or satisfied by contemporary clause
After pop, clause is no longer unit under current top-level assignment
Clause became unit under current top-level assignment
Clause is popped
Clause is popped

Fig. 2. Clause Management in MATHSAT5

by 1) requiring the external solver to implement a specific *SAT worker* interface which defines the communication protocol between the external solver and the rest of MATHSAT5, and 2) requiring the external solver to invoke some *callback functions* in order to notify the rest of MATHSAT5 about specific states in the SAT search.

The SAT worker interface consists of methods for creating SAT variables, adding clauses, propagating literals deduced by the theory solvers, and retrieving the truth values of variables after a Boolean model has been found. In order to work correctly in the context of MATHSAT5, the SAT solver is required to be able to create new variables and add new clauses during search. If it uses some form of preprocessing involving variable elimination, it must also support the ability of freezing some of the variables and correctly handle the addition of clauses containing previously-eliminated variables (see §4.3), or else preprocessing must be turned off.[3] Finally, in order to be usable in an incremental setting, the SAT solver must support solving under assumptions [20] (otherwise, it can only be used for non-incremental queries). In general, implementing such interface amounts to creating a wrapper that invokes the corresponding functions in the API of the SAT solver.[4]

Besides implementing the worker interface, the code of the external SAT solver must also be modified to invoke a number of callback functions provided by MATHSAT5, in order to allow the interaction between the SAT engine and the theory solvers in MATHSAT5 during the SAT search. In particular, the callback functions invoke the theory solvers when either a complete Boolean model or a non-conflicting partial assignment has been found. Invoking the theory solvers allows us to do early pruning, theory consistency checking and the propagation of theory deductions.

In general, the source code of the external SAT solver needs to be patched to include the proper calls to the MATHSAT5 callback functions. However, in our experience the

[3] More generally, all the SAT-based simplification techniques which are not sound in an SMT context (such as e.g. the pure literal rule) must be switched off.

[4] Here, we are implicitly assuming that the SAT solver exposes an API similar to that of modern CDCL solvers such as e.g. MINISAT or LINGELING.

amount of changes required is typically quite small. In our example implementations, using the MiniSat [19] and Cleaneling [17] open-source SAT solvers, the patches consist of less than 150 lines of code.

5 Experimental Evaluation

In this section, we present an experimental evaluation of MathSAT5. We demonstrate two key properties of our solver: first, the improvement over the previous version of MathSAT; second, the usefulness of the new features.

Benchmarks. For our experiments, we use the following classes of benchmarks.

BVuMEM. Benchmarks from the $\mathcal{BV} \cup \mathcal{AR}$ SMT-LIB category. We leave out a family containing only very large but trivial to solve benchmarks.
HSver. Benchmarks originating from practical problems in the verification of hybrid systems. The benchmarks are in the theory of $\mathcal{LA}(\mathbb{Q})$, and represent proof obligations generated by the scenario-based verification algorithms of [12]. Besides the $\mathcal{LA}(\mathbb{Q})$ component, these instances also have a complex Boolean component.
COMP09. Benchmarks from the 2009 SMT-COMP, in the categories entered by MathSAT4 at the time.
LRA11. The application benchmarks of the 2011 SMT-COMP, for the theory of $\mathcal{LA}(\mathbb{Q})$.

The first three classes of benchmarks are considered as non-incremental (i.e. we check satisfiability once per benchmark). The benchmarks in LRA11 are used to test the value of various features of MathSAT5 in an incremental setting[5].

MathSAT Configurations. In our experiments we have used the following versions of MathSAT:

MathSAT4: The latest version of MathSAT4 (version 4.2.17).
MathSAT5: The baseline MathSAT5 configuration.
MathSAT5$_{\text{Preprocessing}}$: MathSAT5 with preprocessing enabled.
MathSAT5$_{\text{Cleaneling}}$: MathSAT5 using Cleaneling as a pluggable SAT solver.
MathSAT$_{\text{MiniSat}}$: MathSAT5 using MiniSat as a pluggable SAT solver.[6]

Experimental Set Up. All benchmarks were run on an xcore X5650 platform running Linux version 2.6.32, with a 32GB memory limit and a 20 minute time limit. In the tables, we use the following acronyms: RT for Runtime, TO for Time Out, MO for Memory Out.

[5] The benchmarks in HSver could be also organized as incremental; however, the number of subsequent satisfiability queries is very low (two orders of magnitude lower than LRA11), and thus the results are not particularly informative.

[6] Notice that, although both MiniSat and Cleaneling support SAT preprocessing, we had to turn it off when integrating them with MathSAT5, since their SAT preprocessing procedures do not satisfy the requirements listed in §4.3 (see also §4.4).

Table 1. Results for MATHSAT5 with and without preprocessing on the BVuMEM and HSver benchmark classes

Benchmark Family	Size	MATHSAT5				MATHSAT5$_{\text{PREPROCESSING}}$			
		#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO
brummayerbiere2	22	15	2218	5	2	**16**	**2014**	**6**	**0**
brummayerbiere	293	229	25698	64	0	**233**	**22620**	**60**	**0**
calc2	36	30	7855	6	0	**30**	**7301**	**6**	**0**
stp	40	26	2659	6	8	**27**	**3127**	**5**	**8**
HSver	279	260	6192	19	0	**279**	**2182**	**0**	**0**

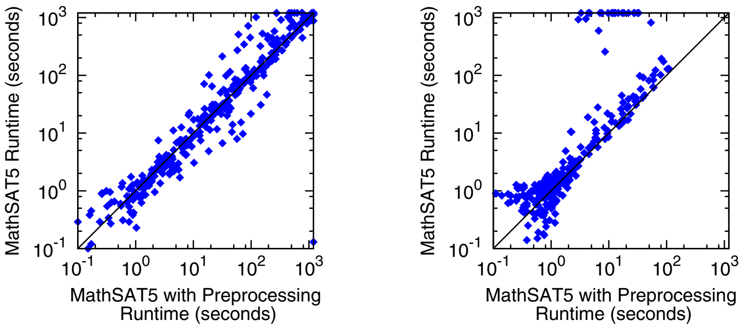


Fig. 3. Impact of preprocessing in the BVuMEM (left) and HSver (right) classes

Experiments. The intent of the first set of experiments is to evaluate the impact of SAT-level preprocessing. We focus on theories that cannot be directly reduced to pure SAT, showing that our approach is useful outside of pure $\mathcal{BV}$ problems. Table 1 shows the results of running MATHSAT5 both with and without preprocessing on the BVuMEM and HSver benchmarks. Figure 3 presents the corresponding scatter plots. In the BVuMEM benchmarks, the activation of the preprocessor allows MATHSAT5 to solve a higher number of instances. We notice that the activation of the preprocessor is not always positive, as it may result in time outs in cases solved without preprocessing (3 benchmarks). In terms of runtime, on the benchmarks solved in both cases, preprocessing yields a 15% In the HSver benchmarks, on the other hand, the positive effect of preprocessing is very evident, with 19 more instances solved, and a 2.8x speed up on average runtime. On single benchmarks, we notice an improvement of up to two orders of magnitude.

In the second set of experiments we compare MATHSAT5 against our previous solver MATHSAT4, using the COMP09 benchmarks. The results of the experiment are aggregated in Table 2, and displayed in scatter plots in Figures 4 and 5. From the data presented we can clearly conclude that MATHSAT5 outperforms MATHSAT4. We notice significant improvements in the $\mathcal{EUF}$ and $\mathcal{LA}(\mathbb{Z})$ categories.

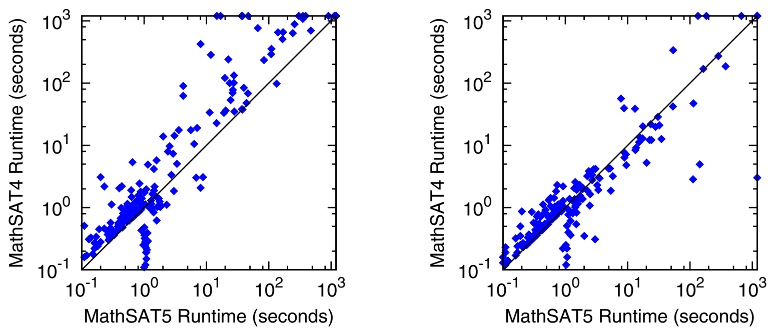


Fig. 4. MATHSAT5 versus MATHSAT4 on $\mathcal{EUF}$ (left) and $\mathcal{LA}(\mathbb{Q})$ (right)

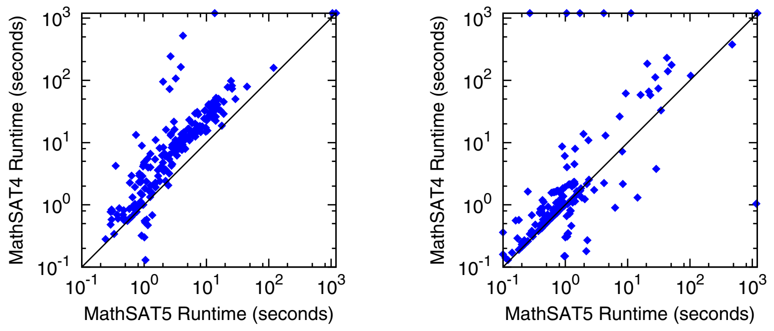


Fig. 5. MATHSAT5 versus MATHSAT4 on $\mathcal{LA}(\mathbb{Z})$ (left) and $\mathcal{BV}$ (right)

Table 2. A comparison between MATHSAT4 and MATHSAT5 on the COMP09 benchmarks

Category	Size	MATHSAT4				MATHSAT5			
		#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO
$\mathcal{BV}$	200	192	1939	8	0	**197**	**2295**	**3**	**0**
$\mathcal{EUF}$	200	186	9317	14	0	**196**	**6232**	**4**	**0**
$\mathcal{LA}(\mathbb{Z})$	205	202	3985	3	0	**204**	**2205**	**1**	**0**
$\mathcal{LA}(\mathbb{Q})$	202	182	1588	20	0	**184**	**2816**	**18**	**0**

In order to assess the pluggable SAT solver feature, we created to two versions of MATHSAT5 by integrating two external solvers[7]: MINISAT [19] and CLEANELING [17]. The cost of the integration turned out to be very moderate. This supports the claim that specialised SAT solver could be integrated and exploited successfully with a low initial effort.

[7] The code for the integration (see §4.4) is available from the web page of MATHSAT5 [29].

Table 3. A comparison between the MathSAT5$_{\text{Cleaneling}}$, MathSAT5$_{\text{MiniSat}}$ and MathSAT5 solvers on the BVuMEM instances

Benchmark Family	Size	MathSAT5$_{\text{Cleaneling}}$				MathSAT5$_{\text{MiniSat}}$				MathSAT5			
		#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO	#Solved	RT (sec)	#TO	#MO
brummayerbiere2	22	12	709	8	2	**15**	**1831**	**5**	**2**	15	2218	5	2
brummayerbiere	293	164	23383	97	29	184	17044	97	12	**229**	**25698**	**64**	**0**
calc2	36	29	5852	7	0	**36**	**4183**	**0**	**0**	30	7855	6	0
stp	40	27	3595	5	8	**29**	**1765**	**3**	**8**	26	2659	6	8

Table 4. A comparison between MathSAT5$_{\text{MiniSat}}$ and MathSAT5 on the LRA11 instances

Benchmark	MathSAT5$_{\text{MiniSat}}$		MathSAT5	
	Reached bound	Runtime (sec)	Reached bound	Runtime (sec)
bmwlin_20_5_1.inter.bmc_k100	**101**	**25**	101	344
fisher_ring_20_3.inter.bmc_k100	**62**	**1200**	55	1200
dist_controller_15_3.inter.bmc_k100	76	1200	**93**	**1200**
rod_30_3.inter.bmc_k100	**101**	**66**	80	1200
fisher_star_20_3.inter.bmc_k100	**101**	**40**	101	367
rod_30_3.inter.ind_k100	27	1200	**45**	**1200**
mwlin_20_5_1.inter.ind_k100	47	1200	**133**	**1200**
fisher_star_20_3.inter.ind_k100	35	1200	**65**	**1200**
fisher_ring_20_3.inter.ind_k100	33	1200	**51**	**1200**
dist_controller_15_3.inter.ind_k100	31	1200	**69**	**1200**

Then, we compared these two solvers on the BVuMEM benchmarks. The results are detailed in Table 3. Compared to the version with our native solver (Table 1), the performance of the version with MiniSat is mixed: MathSAT5$_{\text{MiniSat}}$ performs slightly better on three families, but much worse on the brummayerbiere family. The version with Cleaneling instead is inferior to our native solver. In general, SAT solvers and DPLL($\mathcal{T}$) SAT enumerators might have different requirements, so it's not obvious that a state-of-the-art SAT solver is always the best choice in DPLL($\mathcal{T}$). For example, the rapid restart policy used by modern SAT solvers might not be the best choice in SMT. Rebuilding the assignment stack after a restart is relatively cheap in pure SAT; however, in SMT it can be more expensive, since the theory solvers still need to perform consistency checks and provide deductions.

We also tested the version with pluggable solver on incremental benchmarks. Since Cleaneling does not support solving under assumptions, and thus cannot be used incrementally by MathSAT5, we compared the performance of MathSAT5 and MathSAT5 using MiniSat on the LRA11 benchmarks set (Table 4). These problems are either bounded model checking (where the benchmark name contains "bmc"), or k-induction (name contains "ind") problems. Interestingly, k-inductions checks are much more efficiently solved by pure MathSAT5, while the version using MiniSat handles bounded model checking instances much more efficiently. We are currently investigating the reasons for this difference.

In order to assess the strength of MathSAT5 relative to the current state of the art (e.g. Boolector and Z3) we rely on the results of the 2011 and 2012 SMT-COMP. The version of MathSAT5 presented in this paper is an extension, with new features, of the version which ran in those competitions. Thus the SMT-COMP results are relevant to this version. The competition results show that in non-incremental categories MathSAT5 is generally competitive with other modern SMT solvers. In the incremental categories, it performs extremely well, winning many of them. Thus MathSAT5 achieves its goal of being an efficient incremental solver, that supports a multitude of logics.

6 Applications

MathSAT has been and is currently used in many research and industrial projects.

We have a long-standing collaboration with Intel FV group at Haifa, Israel, within the Intel- and SRC-funded BOWLING, WOLFLING and WOLF projects, in which MathSAT has been used as backend engine for formal verification of RTL designs microcode [22]. In particular, a customized version of MathSAT is currently integrated within the production version of Intel's microcode-verification suite, MicroFormal, and successfully used inside the company [22]. Another application in the verification of RTL is in the ForSyn [21] tool, where MathSAT is the decision procedure used for checking the equivalence between RTL implementations and their high-level descriptions.

MathSAT has been used as a backend in an extended version of the NuSMV model checker, called NuSMV3 [31]. NuSMV3 is a general synchronous extensions to the publicly available NuSMV2, where MathSAT is used as a backend for SMT-based verification techniques. Among these, we mention bounded model checking, k-induction, and predicate abstraction. In these applications, the role of SMT is to provide a high level representation of the transition system. Various functionalities are exploited, including incremental reasoning, unsatisfiable core extraction, and interpolation.

The availability of MathSAT has provided a basis for the extension of NuSMV to deal with analysis of hybrid systems. Hybrid systems are symbolically modeled in a language called HyDI, and specialized forms of verification [13,12] strongly rely on the availability of advanced capabilities of MathSAT. In the setting of hybrid systems verification, MathSAT also supports the analysis of parametric timed automata [15].

The EuRailCheck project, funded by the European Railways Agency, relies on the MathSAT-based requirements analysis capabilities [16].

The underlying verification capabilities provided by NUSMV3 are used in the ESA-funded projects COMPASS [4], AUTOGEF, FAME, and FOREVER, where complex aerospace systems are modeled in terms of hybrid automata.

MathSAT is used as a backend for the analysis of temporal reasoning under uncertainty [11], within applications in the ESA-funded project IRONCAP.

An important class of applications of MathSAT in software model checking. In particular, MathSAT is integrated within the CPAchecker [3] and UFO [1] model checkers for sequential software, and within Kratos [13], a model checker for sequential and threaded software. Within this setting, MathSAT supports the basic model checking steps (interpolation, predicate discovery, localization and post-image computation) by means of interpolation, unsatisfiable core extraction, and AllSMT. More recently, MATHSAT has been used as backend for an IC3-based approach to software model checking [8], and for parametric analysis of threaded programs [14].

7 Conclusions and Future Developments

In this paper we have presented the SMT tool MATHSAT5. In comparison to its predecessor MATHSAT4, substantial improvements have been made: in addition to significant improvements in efficiency, the key changes include extension to more theories, full support for incrementality, an incremental and SMT-aware preprocessor, and support to plug in third-party SAT solvers.

MATHSAT is a long-term project, and its development is ongoing. First, we plan a deeper investigation of SMT-aware preprocessing techniques, with the goal to make them available within a stand-alone functionality, so that to make MATHSAT work also as an effective *formula simplifier*. Second, we plan to investigate and implement *quantifier elimination* techniques for some of the theories of interest. We are also considering to investigate extensions to non-linear arithmetic.

A research direction we are currently pursuing is that of *Optimization Modulo Theories (OMT)*, which leverages SMT solving from *decision* to *optimization* level by finding models that *minimize* some given cost functions. Our previous work has produced variants of MATHSAT able to minimize cost functions on the pseudo-Boolean and $\mathcal{LA}(\mathbb{Q})$ domains respectively [7,32]. Current and future work in this direction includes the porting of the OMT implementations of [7,32] into the official MATHSAT5 version, and extensions to $\mathcal{LA}(\mathbb{Z})$ and $\mathcal{LA}(\mathbb{QZ})$ cost functions.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A Framework for Abstraction-and Interpolation-Based Software Verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 672–678. Springer, Heidelberg (2012)
2. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability, ch. 26. IOS Press (2009)
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. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, Dependability and Performance Analysis of Extended AADL Models. Comput. J. 54(5) (2011)
5. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
6. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: A Comparative Analysis. Annals of Mathematics and Artificial Intelligence 55(1-2) (2009)
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. Cimatti, A., Griggio, A.: Software Model Checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)
9. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. ACM TOCL 12(1) (2010)
10. Cimatti, A., Griggio, A., Sebastiani, R.: Computing Small Unsatisfiable Cores in SAT Modulo Theories. Journal of Artificial Intelligence Research, JAIR 40, 701–728 (2011)
11. Cimatti, A., Micheli, A., Roveri, M.: Solving Temporal Problems Using SMT: Strong Controllability. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 248–264. Springer, Heidelberg (2012)
12. Cimatti, A., Mover, S., Tonetta, S.: SMT-based Scenario Verification for Hybrid Systems. Formal Methods in System Design (2012)
13. Cimatti, A., Narasamdya, I., Roveri, M.: Software Model Checking with Explicit Scheduler and Symbolic Threads. Logical Methods in Computer Science 8(2) (2012)
14. Cimatti, A., Narasamdya, I., Roveri, M.: Verification of Parametric System Designs. In: Proc. FMCAD. FMCAD (2012)
15. Cimatti, A., Palopoli, L., Ramadian, Y.: Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In: IEEE Real-Time Systems Symposium (2008)
16. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Validation of Requirements for Hybrid Systems: a Formal Approach. TOSEM 21(4) (2013)
17. CLEANELING, http://fmv.jku.at/cleaneling/
18. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
19. 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)
20. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
21. ForSyn, http://www.cs.utexas.edu/ sandip/projects/behavioral-synthesis/index.html
22. Franzén, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. In: FMCAD, pp. 121–128 (2010)
23. The GNU Multi Precision Arithmetic Library, http://gmplib.org
24. Goel, A., Krstić, S., Fuchs, A.: Deciding array formulas with frugal axiom instantiation. In: Proceedings of SMT 2008/BPR 2008, pp. 12–17. ACM, New York (2008)
25. Griggio, A.: An Effective SMT Engine for Formal Verification. PhD thesis, DISI - University of Trento (2009)
26. Griggio, A.: Effective word-level interpolation for software verification. In: FMCAD, pp. 28–36. FMCAD Inc. (2011)
27. Griggio, A., Le, T.T.H., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo linear integer arithmetic. Logical Methods in Computer Science 8(3) (2012)
28. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT Techniques for Fast Predicate Abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006)
29. MathSAT 5, http://mathsat.fbk.eu/
30. Nadel, A., Ryvchin, V., Strichman, O.: Preprocessing in Incremental SAT. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 256–269. Springer, Heidelberg (2012)
31. NuSMV 3, https://es.fbk.eu/tools/nusmv3/
32. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ Cost Functions. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 484–498. Springer, Heidelberg (2012)

Formula Preprocessing in MUS Extraction*

Anton Belov[1], Matti Järvisalo[2], and Joao Marques-Silva[1,3]

[1] Complex and Adaptive Systems Laboratory, University College Dublin, Ireland
[2] HIIT & Department of Computer Science, University of Helsinki, Finland
[3] IST/INESC-ID, Lisbon, Portugal

Abstract. Efficient algorithms for extracting minimally unsatisfiable subformulas (MUSes) of Boolean formulas find a wide range of applications in the analysis of systems, e.g., hardware and software bounded model checking. In this paper we study the applicability of preprocessing techniques for Boolean satisfiability (SAT) in the context of MUS extraction. Preprocessing has proven to be extremely important in enabling more efficient SAT solving. Hence the study of the applicability and the effectiveness of preprocessing in MUS extraction is highly relevant. Considering the extraction of both standard and group MUSes, we focus on a number of SAT preprocessing techniques, and formally prove to what extent the techniques can be directly applied in the context of MUS extraction. Furthermore, we develop a generic theoretical framework that captures MUS extraction problems, and enables formalizing conditions for correctness-preserving applications of preprocessing techniques that are not applicable directly. We experimentally evaluate the effect of preprocessing in the context of group MUS extraction.

1 Introduction

Efficient algorithms for extracting minimally unsatisfiable subformulas (MUSes) of Boolean formulas find a wide range of applications in the analysis of systems, e.g., hardware and software bounded model checking. A variety of different approaches to MUS extraction has been proposed, see [5,9,22,21,23,19] for recent examples and [20] for a survey. Typically the state-of-the-art MUS extraction algorithms use Boolean satisfiability (SAT) solvers as NP-oracles for checking the satisfiability of subformulas in an iterative manner.

In recent years, formula preprocessing has emerged as an extremely important technique in enabling efficient SAT solving (see e.g. [6,8,10,7,11,1,12,15]). Thus, in this paper, we study of the applicability and the effectiveness of preprocessing in the context of MUS extraction.

The result of MUS extraction on a preprocessed input formula F' is an MUS M' of F'. However, since preprocessing changes the formula structure by, e.g., removing clauses and removing or adding literals to clauses, M' is, in general, *not* an MUS of F.

* The first and third authors are financially supported by SFI PI grant BEACON (09/IN.1/I2618), and by FCT grants ATTEST (CMU-PT/ELE/0009/2009) and POLARIS (PTDC/EIA-CCO/123051/2010). The second author is financially supported by Academy of Finland (grants 132812 and 251170).

Hence we are faced with the problem of *reconstructing* an MUS of F from M'. Considering the whole MUS extraction process, in order to benefit from preprocessing, this reconstruction must be performed efficiently. However, even guaranteeing correctness (i.e., ensuring that the reconstructed subformulas are actually MUSes) when applying preprocessing becomes non-trivial. This is especially true for the recently introduced problem of *group* (or *high-level*) [18,22] MUS extraction, which is practically a very relevant generalization of the "plain" MUS extraction problem.

Considering the extraction of both standard and group MUSes, we focus on a number of important SAT preprocessing techniques, including *clause elimination procedures* [7,12] such as *subsumption* and *blocked clause elimination* [14], and resolution-based preprocessing techniques (SatElite-style *variable elimination* [6], *self-subsuming resolution*, and *Boolean constraint propagation*). We show formally to what extent the techniques can be directly applied in the context of MUS extraction. It turns out that, especially in the case of group MUS extraction, maintaining correctness under preprocessing needs extra attention. This is further corroborated by the fact that incorrect results produced by some group MUS extractors that applied preprocessing in the special track of the 2011 SAT Competition on group MUS extraction were likely due to incorrect applications of standard SAT preprocessing techniques (see `http://www.satcompetition.org/2011/` for details). We develop a generic theoretical framework based on *labelled CNFs*, which provides a unifying view to variants of MUS extraction problems, and enables formalizing conditions for correctness-preserving applications of preprocessing techniques that are not applicable directly. Additionally, we experimentally evaluate the effect of preprocessing in the context of group MUS extraction.

2 Preliminaries

For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a disjunction of literals and a CNF formula a conjunction of clauses. A clause can be seen as a finite set of literals and a CNF formula as a finite set of clauses. A *unit clause* contains exactly one literal. A clause is a *tautology* if it contains both x and $\bar{x}$ for some variable x. A clause C is *subsumed* by a clause $C' \subset C$ (viewed as sets of literals). A truth assignment for a CNF formula F is a function τ that maps variables in F to $\{0, 1\}$. If $\tau(x) = v$, then $\tau(\bar{x}) = \bar{v}$, where $\bar{1} = 0$ and $\bar{0} = 1$. A clause C is satisfied by τ if $\tau(l) = 1$ for some $l \in C$. An assignment τ satisfies F if it satisfies every clause in F. A CNF formula is *satisfiable* if there is an assignment that satisfies it, and *unsatisfiable* otherwise. We denote the set of all unsatisfiable and satisfiable CNF formulas, resp., by UNSAT and SAT, resp. Two CNF formulas F and F' are *equisatisfiable* if we have that $F \in \mathsf{SAT}$ iff $F' \in \mathsf{SAT}$.

Minimal Unsatisfiability. A CNF formula F is *minimally unsatisfiable* if (i) $F \in \mathsf{UNSAT}$, and (ii) for any clause $C \in F$, $F \setminus \{C\} \in \mathsf{SAT}$. We denote the set of minimally unsatisfiable CNF formulas by MU. A CNF formula F' is a *minimally unsatisfiable subformula (MUS)* of a formula F if $F' \subseteq F$ and $F' \in \mathsf{MU}$. The set of MUSes of a CNF formula F is denoted by $\mathsf{MUS}(F)$. In general, a given unsatisfiable formula F may contain more than one MUS.

Motivated by several industrially relevant applications, minimal unsatisfiability and related concepts have been extended to CNF formulas where clauses are partitioned into disjoint sets called *groups* [18,22].

Definition 1. *Given an explicitly partitioned unsatisfiable CNF formula $G = G_0 \cup G_1 \cup \cdots \cup G_k$ (a* group MUS instance *or* group CNF formula*), where the G_i's are pair-wise disjoint sets of clauses called* groups*, a group MUS of G is a subset $\mathcal{G}$ of $\{G_1, \ldots, G_k\}$ such that (i) $G_0 \cup \bigcup \mathcal{G}$ (seen as a monolithic CNF formula) is unsatisfiable, and (ii) for any group $G \in \mathcal{G}$, $G_0 \cup \bigcup \left(\mathcal{G} \setminus \{G\}\right)$ is satisfiable. The set of group MUSes of a group MUS instance G is denoted by* $\mathsf{GMUS}(G)$.

Clause Elimination Procedures. Given a CNF formula F, a *clause elimination procedure E* is an algorithm that on input F returns a CNF formula $E(F) \subseteq F$ that is equisatisfiable with F. A specific clause elimination procedure E removes clauses satisfying a *specific (typically polynomial-time computable) redundancy property P_E* from F until fixpoint. In other words, E on input F modifies F by repeating the following: if there is a clause $C \in F$ satisfying P_E, let $F := F \setminus \{C\}$.

An example of a clause elimination procedure is *blocked clause elimination* (BCE), which removes so-called *blocked clauses* [16] from CNF formulas until fixpoint. A literal l in a clause C of a CNF formula F blocks C (with respect to F) if for every clause $C' \in F$ with $\bar{l} \in C'$, the resolvent $(C \setminus \{l\}) \cup (C' \setminus \{\bar{l}\})$ obtained from resolving C and C' on l is a tautology. A clause is blocked (with respect to a fixed CNF formula) if it has a literal that blocks it. Note that clauses that contain pure literals are blocked [14]. Additional well-known clause elimination procedures include *tautology elimination* (removing tautological clauses) and *subsumption elimination* (removing subsumed clauses). These and other more involved clause elimination procedures are analyzed in the context of CNF satisfiability in [12,13].

Resolution-Based Preprocessing. The resolution rule states that, given two clauses $C_1 = (l \vee A)$ and $C_2 = (\bar{l} \vee B)$, the implied clause $C = (A \vee B)$, called the *resolvent* of C_1 and C_2, can be inferred by *resolving* on the literal l. We write $C = C_1 \otimes_l C_2$. This is lifted to two sets S_l and $S_{\bar{l}}$ of clauses that all contain the literal l and $\bar{l}$, resp., by $S_l \otimes_l S_{\bar{l}} = \{C_1 \otimes_l C_2 \mid C_1 \in S_l, C_2 \in S_{\bar{l}},$ and $C_1 \otimes_l C_2$ is not a tautology$\}$.

Variable Elimination (VE) [6] is defined following the Davis-Putnam procedure (DP). The elimination of a variable x in the whole CNF can be computed by pair-wise resolving each clause in S_x with every clause in $S_{\bar{x}}$. Replacing the original clauses in $S_x \cup S_{\bar{x}}$ with the set of *non-tautological* resolvents $S = S_x \otimes_x S_{\bar{x}}$ gives the CNF $(F \setminus (S_x \cup S_{\bar{x}})) \cup S$ which is equisatisfiable with F. In order to avoid exponential worst-case space complexity, VE is bounded typically as follows: a variable x is allowed to be eliminated only if $|S| \leq |S_x \cup S_{\bar{x}}| + \Delta$, i.e., the resulting CNF formula $(F \setminus (S_x \cup S_{\bar{x}})) \cup S$ will not contain more than a constant Δ more clauses than the original formula F (typically $\Delta = 0$ [6]). VE is currently one of the most important SAT preprocessing techniques, as witnessed by e.g. the SatElite preprocessor [6].

In the following, we will consider individual steps of variable elimination. Given a CNF formula F, the result of eliminating the variable x from F is $\mathsf{VE}(F, x) = (F \setminus (S_x \cup S_{\bar{x}})) \cup (S_x \otimes_x S_{\bar{x}})$. Note that in the case x appears in one polarity only (i.e., x is a pure literal), this operation simply removes all clauses that contain x.

Unit propagation Given a CNF formula F, unit propagation on F refers to applying the following steps on F until fixpoint: if there is a unit clause (l) in F, remove all clauses with the literal l from F, and remove the literal $\bar{l}$ from all clauses in F. We will consider individual steps of unit propagation on a CNF formula F, where a single literal l is propagated: $\mathsf{BCP}(F, l) = \{C \in F \mid C \cap \{l, \bar{l}\} = \emptyset\} \cup \{C \setminus \{\bar{l}\} \mid C \in F,\ \bar{l} \in C\}$.

Self-Subsuming Resolution (SSR) Given a CNF formula F, the self-subsuming resolution rule states that, given two clauses $C, D \in F$ such that $l \in C$ and $\bar{l} \in D$ for some literal l, and D is subsumed by $C \otimes_l D$, D can be *replaced* with $C \otimes_l D$ in F (or, informally, $\bar{l}$ can be removed from D). Hence a step of self-subsuming resolution, resolving C and D on l, results in the formula $\mathsf{SSR}(F, C, D, l) = (F \setminus D) \cup \{C \otimes_l D\}$. Regarding the practical importance of SSR, as noted in [6], applying SSR in combination with VE and subsumption elimination can give notable improvements w.r.t. applying VE alone.

3 Direct Preprocessing in MUS Extraction

In this section we address the question of the *direct* applicability of CNF preprocessing techniques described in Sect. 2 in the context of MUS extraction. That is, whether we can simply apply a technique to a formula F (keeping track of the changes), extract an MUS of the preprocessed formula, and reconstruct an MUS of F from it in an efficient and natural way.

3.1 Clause Elimination Procedures

Plain MUS Extraction It is rather straightforward to observe that clause elimination procedures can be directly applied in the context of plain MUS extraction: for any MUS M of a CNF formula F', such that F' is the result of applying any combination of clause elimination procedures on an input CNF F, we have that M is an MUS of F.

Proposition 1. *If F' is a result of applications of clause elimination procedures to an unsatisfiable CNF formula F, then* $\mathsf{MUS}(F') \subseteq \mathsf{MUS}(F)$.

Proof. Since $F' \subseteq F$, we have $M \subseteq F$ for any $M \in \mathsf{MUS}(F')$. Furthermore, since $M \in \mathsf{MU}$, we have $M \in \mathsf{MUS}(F)$. $\square$

Note the inclusion instead of equality in Proposition 1: consider $F = F_1 \cup F_2$ such that $F_1, F_2 \in \mathsf{MU}$, $F_1 \cap F_2 = \emptyset$. Since both F_1 and F_2 are unsatisfiable on their own, there is a clause elimination procedure that removes all clauses either in F_1 or F_2 from F.

Group MUS Extraction. We say that a clause elimination procedure S is applied on a group MUS instance $G = \{G_0, G_1, \ldots, G_n\}$ when referring to applying S on G seen as the monolithic CNF formula $\bigcup_{i=0}^{n} G_i$. The resulting group MUS instance is $S(G) = \{G'_0, G'_1, \ldots, G'_n\}$, where for each $i = 0..n$ we have $G'_i = G_i \cap S(\bigcup_{i=0}^{n} G_i)$. A natural idea for reconstructing a GMUS M of G from a GMUS M' of $S(G)$ would be to consider $M = \{G_i \in G \mid G'_i \in M'\}$. However, we will show that this natural idea does not generally work: whether M is always guaranteed to be a GMUS of G depends critically on the choice of the clause elimination procedure S. Surprisingly, even subsumption elimination is problematic, as witnessed by the following example.

Example 1. Consider the group MUS instance $G = \{G_0, G_1, G_2\}$, where $G_0 = \{(\bar{r})\}$, $G_1 = \{(p \vee q), (\bar{q} \vee r), (\bar{p} \vee r)\}$, and $G_2 = \{(p)\}$. Here $\{G_1\}$ is the only group MUS. Here $(p) \in G_2$ subsumes $(p \vee q) \in G_1$. However, $\{G'_1\}$ with $G'_1 = G_1 \setminus \{(p \vee q)\}$ is not a group MUS of $\{G'_0 = G_0, G'_1, G'_2 = G_2\}$ since $G_0 \cup G'_1 \in \mathsf{SAT}$; $\{G'_1, G'_2\}$ is. ■

A similar proposition to that of Proposition 1 in the context of group MUS extraction can be shown for the restricted case of what we call "monotonic" clause elimination procedures: a clause elimination procedure S is monotonic if, for any two CNFs F and F' s.t. $F' \subseteq F$, we have that $S(F') \subseteq S(F)$.

Proposition 2. *Let $G = \{G_0, G_1, \ldots, G_k\}$ be a group MUS instance, and S any monotonic clause elimination procedure. For any GMUS $M' \subseteq S(G)$ of the group MUS instance $S(G) = \{G'_0, G'_1, \ldots, G'_k\}$ obtained from applying S on G, $M = \{G_i \in G \mid G'_i \in M'\}$ is a GMUS of G.*

Proof. Assume that M is not a group MUS of G. Take any group MUS $M'' \subset M$ of M. The monolithic CNF formula $S(M'')$ is unsatisfiable. Since S is monotonic, for each group, say G''_i, in $S(M'')$, M' contains a group G'_i that is a superset of G''_i. Furthermore, since $M'' \subset M$, there is a group in M' that is not a superset of any group in $S(M'')$. It follows that M' is not a group MUS of G', which is a contradiction. □

In other words, any monotonic clause elimination procedure can be safely used for preprocessing in the context of plain MUS and group MUS extraction. In addition to tautology elimination, this includes, e.g., blocked clause elimination.

Proposition 3. *Let $G = \{G_0, G_1, \ldots, G_k\}$ be a group MUS instance. For any GMUS $M' \subseteq \mathsf{BCE}(G)$ of the group MUS instance $\mathsf{BCE}(G) = \{G'_0, G'_1, \ldots, G'_k\}$ obtained from applying BCE on G, $M = \{G_i \in G \mid G'_i \in M'\}$ is a GMUS of G.*

Proof. By Proposition 2, it is enough to show that BCE is monotonic. Recall that a literal l in a clause C of a CNF formula F blocks C (with respect to F) if for *every* clause $C' \in F$ with $\bar{l} \in C'$, the resolvent $(C \setminus \{l\}) \cup (C' \setminus \{\bar{l}\})$ is a tautology. Note that, in particular, l blocks C if $\bar{l}$ does not appear in any clause of F (i.e. l is pure). Hence, if l blocks C wrt F, then l blocks C wrt any $F' \subseteq F$, and thus $\mathsf{BCE}(F') \subseteq \mathsf{BCE}(F)$. □

Furthermore, *pure literal elimination* (PLE) is also covered. The CNF formula $\mathsf{PLE}(F)$ resulting from applying pure literal elimination on F is the formula at the fixpoint of the following: while there is a pure literal l in F, let $F := F \setminus \{C \mid l \in C\}$.

Proposition 4. *Let $G = \{G_0, G_1, \ldots, G_k\}$ be a group MUS instance. For any GMUS $M' \subseteq \mathsf{PLE}(G)$ of the group MUS instance $\mathsf{PLE}(G) = \{G'_0, G'_1, \ldots, G'_k\}$ obtained from applying PLE on G, $M = \{G_i \in G \mid G'_i \in M'\}$ is a GMUS of G.*

Proof. By Proposition 2, since PLE is clearly monotonic. □

Notice that any monotonic clause elimination procedure is also confluent (i.e., has a unique fixpoint). However, the opposite does not hold: a counterexample is subsumption elimination, which is confluent but not monotonic (recall Example 1).

3.2 Resolution-Based Preprocessing

Unit propagation

Plain MUS Extraction For the following, given a CNF formula F, let $F' = \mathsf{BCP}(F, l)$ where $(l) \in F$. For a clause C in F', the BCP *support* $\mathsf{support}_{\mathsf{BCP}}(C, F)$ of C in F is $\{C\}$ if $C \in F$, and $\{(l), (\bar{l} \vee C)\}$ (the premises that produced C) otherwise. A natural idea for reconstructing an MUS M of F from an MUS M' of F' would be to let $M = \bigcup_{C \in M'} \mathsf{support}_{\mathsf{BCP}}(C, F)$. Indeed, in the context of plain MUS extraction, this natural idea works, i.e., M is always guaranteed to be an MUS of F.

Proposition 5. *Let M' be an MUS of $F' = \mathsf{BCP}(F, l)$, where $(l) \in F$. Let $M = \bigcup_{C \in M'} \mathsf{support}_{\mathsf{BCP}}(C, F)$. Then $M \in \mathsf{MUS}(F)$.*

Proof. Assume w.l.o.g. that we have $F = \{(l), (l \vee C_1'), \dots, (l \vee C_n'), (\bar{l} \vee C_1), \dots, (\bar{l} \vee C_m)\} \cup R$, where R is the set of clauses in F which do not contain the variable of l. Hence the formula $F' = \mathsf{BCP}(F, l) = \{C_1, \dots, C_m\} \cup R$. We have $M' \in \mathsf{MU}$, and want to show $M \in \mathsf{MU}$. Note that if $M' \subseteq R$, then $M = M'$, and we are done.

Otherwise, let $M' = \{C_{i_1}, \dots, C_{i_k}\} \cup R'$, where $C_{i_j} \in \{C_1, \dots, C_m\}$, $C_{i_j} \notin R'$, and $R' \subseteq R$. Then, we have $M = \{(l), (\bar{l} \vee C_{i_1}), \dots, (\bar{l} \vee C_{i_k})\} \cup R'$. Clearly $\mathsf{BCP}(M, l) = M'$, and since $M' \in \mathsf{UNSAT}$, M must also be UNSAT (by the soundness of BCP). Let now C' be any clause in M, and let $\hat{M} = M \setminus \{C'\}$. If $C' \neq (l)$, then $\mathsf{BCP}(\hat{M}, l) \subset M'$, and since $M' \in \mathsf{MU}$, we have $\mathsf{BCP}(\hat{M}, l) \in \mathsf{SAT}$, and so, by the soundness of BCP, $\hat{M} \in \mathsf{SAT}$. If $C' = (l)$, then $\hat{M} = \{(\bar{l} \vee C_{i_1}), \dots, (\bar{l} \vee C_{i_k})\} \cup R'$. But, since $M' \in \mathsf{MU}$, we have $R' \in \mathsf{SAT}$. Furthermore, the variable of l does not appear in R', and so setting l to 0 will satisfy the rest of the clauses in $\hat{M}$, and so $\hat{M} \in \mathsf{SAT}$.

We conclude that $M \in \mathsf{UNSAT}$, and, for any $C' \in M$, $M \setminus \{C'\} \in \mathsf{SAT}$. Hence, $M \in \mathsf{MU}$, and since $M \subset F$, we have $M \in \mathsf{MUS}(F)$. □

In other words, if a formula F_n is the result of an application of a sequence of BCP steps $F_2 = \mathsf{BCP}(F_1, l_1)$, ..., $F_n = \mathsf{BCP}(F_{n-1}, l_{n-1})$, to a formula F_1, then given an MUS M_n of F_n, we can reconstruct an MUS M_1 of F_1 by taking the *transitive support* of the clauses in M_n, i.e., $M_{n-1} = \bigcup_{C \in M_n} \mathsf{support}_{\mathsf{BCP}}(C, F_{n-1})$, ..., $M_1 = \bigcup_{C \in M_2} \mathsf{support}_{\mathsf{BCP}}(C, F_2)$. In particular, if $\emptyset \in F_n$, i.e. if the sequence of BCP steps results in a conflict, then the clauses of F_1 that were used to derive the conflict constitute an MUS of F_1. Thus Proposition 5 is a generalization of [17, Proposition 1] that states that inconsistent subformulas detected by unit propagation are minimally unsatisfiable.

Group MUS Extraction. In the context of group MUS extraction, however, unit propagation *cannot* be safely applied over different groups by simply applying BCP on the monolithic CNF formula $F_G = G_0 \cup \dots \cup G_n$, where $G = \{G_0, \dots, G_n\}$ is the input group MUS instance. An intrinsic problem arises from the fact that $\mathsf{BCP}(F, l)$ can be seen as the combination of elimination of all clauses that are subsumed by (l) in F, and $\mathsf{VE}(F', l)$, where F' is the CNF formula resulting from the subsumption elimination step w.r.t. (l). More concretely, recall Example 1 which applies naturally to BCP as well. Another intrinsic problem in applying BCP steps using clauses from different groups is that the resolvents would inherit multiple group identities. Additionally, the sets of inherited group identities is dependent on the BCP variable ordering, as shown next.

Example 2. Consider the group MUS instance $G = \{G_0, \ldots, G_3\}$ with $G_0 = \{(x), (y)\}$, $G_1 = \{(z \vee p \vee q)\}$, $G_2 = \{(\bar{x} \vee \bar{z})\}$, and $G_3 = \{(\bar{y} \vee \bar{z}), (p \vee \bar{q}), (\bar{p} \vee q), (\bar{p} \vee \bar{q})\}$. Here $\{G_1, G_3\}$ is a group MUS of G. Assume now that a sequence of BCP steps is applied to G, viewed as a monolithic CNF formula $F_G = G_0 \cup \cdots \cup G_3$. There are two possible BCP sequences: both sequences produce an unsatisfiable CNF formula that contains all four binary clauses over p and q.

Now consider the possible supports of the clause $C = (p \vee q)$. If the first step is $\mathsf{BCP}(F_G, x)$, then the transitive support of C in F_G is $\{(x), (\bar{x} \vee \bar{z}), (z \vee p \vee q)\}$. In this case the derivation of C involves clauses from G_0, G_1, and G_2. If the first step is $\mathsf{BCP}(F_G, y)$, then the transitive support of C in F_g is $\{(y), (\bar{y} \vee \bar{z}), (z \vee p \vee q)\}$, involving clauses from G_0, G_1, G_3. Now, if we would associate C with all groups in its support, in the former case starting with $\mathsf{BCP}(F_G, x)$ (i.e., under a variable ordering preferring x to y) we end up with $\{G_1, G_2, G_3\} \supset \{G_1, G_3\}$. ■

A partial way of safely applying BCP on a group MUS instance $G = \{G_0, G_1, \ldots, G_k\}$ is to apply BCP fully on the special group G_0. In case unit propagation on G_0 alone leads to a conflict, then G has a single group MUS, namely the empty set. Otherwise, the derived unit clause can be propagated *individually* inside each group G_i, $1 \leq i \leq k$. The intuitive justification for this solution is that in the instance preprocessed with BCP, the transitive support of any clause $C \in G_i$ consists only of clauses of G_0 and a single G_i. By definition, the clauses in G_0 are always included in the unsatisfiability check for any selection of groups G_i, where $i > 0$, and furthermore, this way the group identities will not mix between the other groups.

Self-Subsuming Resolution

While the support for $\mathsf{BCP}(F, l)$ allows to reconstruct a plain MUS of F from an MUS of $\mathsf{BCP}(F, l)$, this technique fails for SSR under the following natural definition of support: given a CNF formula F, let $F' = \mathsf{SSR}(F, C, D, l)$. For a clause E in F', the SSR *support* $\mathsf{support}_{\mathsf{SSR}}(E, F)$ of E in F is $\{E\}$ if $E \in F$, and $\{C, D\}$ otherwise. As with the case of BCP support, this definition allows to recover the resolution step involved in the procedure. Consider the following example.

Example 3. Consider the CNF formula $F = \{(\bar{x} \vee p), (x \vee p \vee q), (\bar{p}), (x \vee \bar{q}), (\bar{x})\}$. After the application of self-subsuming resolution to the first two clauses of F we obtain the formula $F' = \{(\bar{x} \vee p), (p \vee q), (\bar{p}), (x \vee \bar{q}), (\bar{x})\}$. The only MUS of F' is $M' = \{(p \vee q), (\bar{p}), (x \vee \bar{q}), (\bar{x})\}$. Since $\mathsf{support}_{\mathsf{SSR}}((p \vee q), F) = \{(\bar{x} \vee p), (x \vee p \vee q)\}$, the union of the supports of all clauses in M' is precisely the formula F, which can easily be seen to *not* be in MU. ■

Variable Elimination

Since unit propagation is a special case of variable elimination, the problems discussed above with direct applications of BCP on the group MUS level apply to VE as well. However, similarly as for SSR, VE is problematic even in the context of *plain* MUS extraction. Intuitively, part of the problem is that the resolvents produced by a step $\mathsf{VE}(F, x)$ of variable elimination can have multiple pairs of supports, i.e., are produced via more than one distinct pair of premises (note that this is not the case for BCP). The problems caused by this behaviour are highlighted by the following example.

Example 4. Consider the CNF formula $F = A \cup R$, where
$A = \{(x \vee p \vee q \vee r), (x \vee \bar{q} \vee r), (\bar{x} \vee p \vee s), (\bar{s}), (\bar{x} \vee q)\}$ and
$R = \{(p \vee q \vee \bar{r}), (p \vee \bar{q} \vee \bar{r}), (\bar{p} \vee q \vee r), (\bar{p} \vee q \vee \bar{r}), (\bar{p} \vee \bar{q} \vee r), (\bar{p} \vee \bar{q} \vee \bar{r})\}$.
Notice that R is the set of all possible clauses on p, q, r, except $(p \vee q \vee r)$ and $(p \vee \bar{q} \vee r)$. Then, $F' = \mathsf{VE}(F, x) = A' \cup R$, where $A' = \{(p \vee q \vee r \vee s), (p \vee q \vee r), (p \vee \bar{q} \vee r \vee s), (\bar{s})\}$. F' has two MUSes: $M_1 = \{(p \vee q \vee r \vee s), (p \vee \bar{q} \vee r \vee s), (\bar{s})\} \cup R$ and $M_2 = \{(p \vee q \vee r), (p \vee \bar{q} \vee r \vee s), (\bar{s})\} \cup R$. Consider the idea of computing a *minimal support* of M_1 and M_2, i.e., the minimal set of premises $P \subseteq F$ such that $M_1 \subseteq \mathsf{VE}(P, x)$ and $M_2 \subseteq \mathsf{VE}(P, x)$, respectively, with the idea that such a minimal support would be an MUS of F. The minimal supports of M_1 and M_2 are $\{(x \vee p \vee q \vee r), (x \vee \bar{q} \vee r), (\bar{x} \vee p \vee s), (\bar{s})\} \cup R$ and $\{(x \vee p \vee q \vee r), (x \vee \bar{q} \vee r), (\bar{x} \vee p \vee s), (\bar{s}), (\bar{x} \vee q)\} \cup R = A \cup R$, respectively. The former is an MUS of F. However, the latter is not; in other words, even taking such a restricted, and "tightened-up", version of support for reconstructing an MUS is not generally correct. ■

For enabling direct applications of VE on group MUS instances, VE needs to be restricted. As in the case of BCP, VE can be applied solely on G_0, seen as a CNF formula, replacing the original G_0 with the resulting formula in the original instance. Furthermore, correctness is preserved if VE is applied *inside each group* G_i, $1 \leq i \leq k$, meaning that "internal" variables that occur only in clauses of a single group can be eliminated.

However, compared to such "ad hoc" technique-specific restrictions for applying preprocessing techniques in the context of group MUS extraction, a more generic framework for guaranteed correctness-preserving applications for different preprocessing techniques is called for. In the next two sections, we develop such a framework based on the concept of so-called *labelled CNF formulas* [2]. We then formally prove correctness of labelled variants of clause elimination and resolution-based preprocessing techniques for MUS extraction problems expressed in terms of labelled CNF formulas.

4 Labelled CNF Formulas

Assume a countable set of labels $Lbls$. A *labelled clause* (L-clause) is a tuple $\langle C, L \rangle$, where C is a clause, and L is a finite (possibly empty) subset of $Lbls$. We denote the label-sets by superscripts, i.e. C^L is the labelled clause $\langle C, L \rangle$. A *labelled CNF (LCNF)* formula is a finite set of labelled clauses. For an LCNF formula Φ, let $Cls(\Phi) = \bigcup_{C^L \in \Phi} \{C\}$ be the *clause-set* of Φ, and $Lbls(\Phi) = \bigcup_{C^L \in \Phi} L$ be the *label-set* of Φ. LCNF satisfiability is defined in terms of the satisfiability of the clause-sets of an LCNF formula: Φ is satisfiable if and only if $Cls(\Phi)$ is satisfiable. We will re-use the notation SAT (resp. UNSAT) for the set of satisfiable (resp. unsatisfiable) LCNF formulas[1]. However, the semantics of minimal unsatisfiability and MUSes of labelled CNFs are defined in terms of their label-sets via the concept of the *induced subformula*.

Definition 2 (Induced subformula). *Let Φ be an LCNF formula, and let $M \subseteq Lbls(\Phi)$. The subformula of Φ* induced by *M is the LCNF formula $\Phi|_M = \{C^L \in \Phi \mid L \subseteq M\}$.*

[1] To avoid overly optimistic complexity results, we will tacitly assume that the sizes of label-sets of the clauses in LCNFs are polynomial in the number of the clauses.

In other words, $\Phi|_M$ consists of those labelled clauses of Φ whose label-sets are included in M, and so $Lbls(\Phi|_M) \subseteq M$, and $Cls(\Phi|_M) \subseteq Cls(\Phi)$. Alternatively, any clause that has at least one label outside of M is removed from Φ. Thus, it is convenient to talk about the *removal* of a label from Φ. Let $l \in Lbls(\Phi)$ be any label. The LCNF formula $\Phi|_{M\setminus\{l\}}$ is said to be obtained by the *removal of label l from* Φ.

Definition 3 (Minimally Unsatisfiable LCNF). *An LCNF formula Φ is* minimally unsatisfiable *(denoted $\Phi \in \mathsf{LMU}$) if $\Phi \in \mathsf{UNSAT}$, and for all $M \subset Lbls(\Phi)$, $\Phi|_M \in \mathsf{SAT}$.*

Definition 4 (Labelled MUS). *Let Φ be an LCNF formula. A set of labels $M \subseteq Lbls(\Phi)$ is a* labelled MUS (LMUS) *of Φ ($M \in \mathsf{LMUS}(\Phi)$), if $\Phi|_M \in \mathsf{LMU}$.*

Note that LMUSes are sets of *labels*, rather than sets of clauses; this is motivated by the following example. Also, note that the empty set can be an LMUS of Φ (this is the case when the subset of clauses of $Cls(\Phi)$ labelled with $\emptyset$ is unsatisfiable), and for any LMUS M of Φ, $Cls(\Phi|_M)$ includes *all* clauses of Φ labelled with $\emptyset$. Finally, if $M \in \mathsf{LMUS}(\Phi)$, then $Lbls(\Phi|_M) = M$ (note the equality). We now illustrate how some of the notions of minimal unsatisfiability get represented in the framework of LCNFs.

Example 5. (a) Let $F = \{C_1, \dots, C_n\}$ be a CNF formula, and let $\{i\}$ be the label-set of clause C_i. For any LMUS M of $\Phi = \{C_i^{\{i\}} \mid C_i \in F\}$, the CNF formula $\{C_i \mid i \in M\}$ is an MUS of F (and vice versa). (Notice that this is a reduction from MU to LMU.)
(b) Let $F = G_0 \cup G_1 \cup \dots G_k$ be a group CNF formula. For each $C \in F$, take the label-set of C to be $\emptyset$ if $C \in G_0$, and $\{i\}$ if $C \in G_i$ for $i \geq 1$. For any LMUS M of the resulting LCNF Φ, $\{G_i \mid i \in M\}$ is a group MUS of F (and vice versa).
(c) For a CNF formula F and $C \in F$, let the set of variables of C be the label-set of C. Any LMUS M of the resulting LCNF is a *variable-MUS* [4] of F (and vice versa). ■

In the following, we refer to the LCNF formula constructed from a CNF formula F as in Example 5(a) as the LCNF *associated* with F; similarly, the LCNF formula constructed from the group CNF formula F as in Example 5(b) is referred to as the LCNF associated with the group CNF F. Notice that in Example 5(c) the label-sets of clauses are not necessarily disjoint. This allows to capture the semantics of "intersecting" groups, or, to put it differently, the multiple group identity of clauses (recall the discussion of BCP in the context of group MUS extraction in Section 3).

Computing LMUSes

It is not difficult to see that the LMUS extraction problem can be reduced to the group MUS extraction problem: given an LCNF formula Φ. For each label $l \in Lbls(\Phi)$, introduce a fresh variable p_l. For each L-clause $C^L \in \Phi$, create the clause $C \vee \bigvee_{l \in L} p_l$, and put the resulting clauses into the group G_0. Finally, for each $l \in Lbls(\Phi)$ create a singleton group $G_l = \{(\bar{p}_l)\}$. The resulting group-CNF formula $F_\Phi = \{G_0\} \cup \{G_l \mid l \in Lbls(\Phi)\}$ is equisatisfiable with Φ. Furthermore, $\{G_{l_1}, \dots, G_{l_k}\}$ is a group-MUS of F_Φ if and only if $\{l_1, \dots, l_k\}$ is an LMUS of Φ. We omit the proof, but the argument relies on the fact that a removal of a group G_l from F_Φ leaves the literal p_l pure in the clauses of G_0, thus satisfying all clauses with p_l. This in turn is equivalent to the removal of all clauses $C^L \in \Phi$ with $l \in L$, i.e., the removal of the label l from Φ. Note that this reduction together with Example 5(a) can be used to show that the LMU decision problem is DP-complete.

Although the reduction from LMUS extraction to group MUS extraction enables the use of any group MUS extractor for the computation of LMUSes, we observe that in fact there is a simpler and likely more efficient way to compute LMUSes: one can simply load the clauses of the group G_0 of the formula F_Φ into an incremental SAT solver (such as Minisat), and use the variables p_l as *assumption variables*. Notice that the state-of-the-art assumption-based MUS extractors, such as MUSer2 [3] which is used in the experiments of this work, already do *exactly* this when computing MUSes and group-MUSes.

With this practical motivation, we will next provide liftings of the "problematic" preprocessing techniques (recall Section 3) to the labelled MUS setting. The liftings resolve the problems discussed in Section 3, and are safe to implement and apply using assumption variables.

5 Preprocessing in LMUS Extraction

We proceed by lifting clause elimination and resolution-based preprocessing techniques to the labelled case, resulting in correctness-preserving preprocessing techniques for labelled CNFs that are applicable in the general setting of group MUS extraction. It should be noted that labelled CNFs can be used to generalize all concepts related to minimal unsatisfiability and irredundancy (e.g. MSSes, MESes, MaxSAT, etc.) in various settings (clauses, groups, variables, circuits, etc.) [2]. As a by-product, given the natural mapping between plain and group MUS instances described in Example 5, this opens a path for correctness-preserving preprocessing for these settings as well.

5.1 Labelled Clause Elimination

While monotonic clause elimination procedures, including blocked clause elimination, can be directly applied in the group MUS context (recall Proposition 2), for other clause elimination procedures direct applicability appears to be limited. Especially, subsumption elimination cannot be directly applied (recall Example 1).

A correctness-preserving lifting of clause elimination procedures which preserve *logical equivalence* to the general setting of LMUS extraction is provided by the following proposition. Note that subsumption elimination is one of such procedures.

Proposition 6. *Let Φ be an LCNF formula such that for some clauses $C_1^{L_1}, \ldots, C_k^{L_k}$ and C^L in Φ, $\{C_1, \ldots, C_k\} \models C$ and $\bigcup_{1 \le i \le k} L_i \subseteq L$. Then, any LMUS of $\Phi \setminus \{C^L\}$ is an LMUS of Φ.*

Proof. Let $\Phi' = \Phi \setminus \{C^L\}$, and let M be an LMUS of Φ', i.e. $\Phi'|_M \in \mathsf{LMU}$. We need to show that $\Phi|_M \in \mathsf{LMU}$. Note that since $\Phi = \Phi' \cup \{C^L\}$ we have $\Phi|_M = \Phi'|_M \cup \{C^L\}|_M$. Thus, if $L \nsubseteq M$, then $\Phi|_M = \Phi'|_M$, and we are done since $\Phi'|_M \in \mathsf{LMU}$.

If $L \subseteq M$, then $C^L \in \Phi|_M$, and since $\bigcup_{1 \le i \le k} L_i \subseteq L$, *all* clauses $C_i^{L_i}$ are in $\Phi|_M$, and hence in $\Phi'|_M$. Consider any label $l \in M$, and let $M' = M \setminus \{l\}$. If $l \in L$, then

$C^L \notin \Phi|_{M'}$, and therefore $\Phi|_{M'} = \Phi'|_{M'} \in \mathsf{SAT}$, since $\Phi'|_M \in \mathsf{LMU}$. If $l \notin L$, then $\Phi|_{M'} = \Phi'|_{M'} \cup \{C^L\}$, and since $\bigcup L_i \subseteq L$, all clauses $C_i^{L_i}$ are in $\Phi'|'_M$. Since $\Phi'|_{M'} \in \mathsf{SAT}$, any of its satisfying assignments satisfies all clauses $C_i^{L_i}$, and also the clause C since $\{C_1, \ldots, C_k\} \models C$. Hence $\Phi|_{M'} \in \mathsf{SAT}$, and for any $l \in M$, $\Phi|_{M \setminus \{l\}} \in \mathsf{SAT}$, and so $\Phi|_M \in \mathsf{LMU}$. □

Applying Proposition 6 to subsumption elimination, we obtain that a clause $C_1^{L_1}$ subsumed by clause $C_2^{L_2}$ can be eliminated correctly (w.r.t. to LMUS computation) if $L_2 \subseteq L_1$. In particular, in the group-MUS setting, all clauses subsumed by the clauses from the same group, or by the clause from group G_0, can be eliminated safely.

5.2 Labelled Resolution-Based Preprocessing

We now introduce liftings of the resolution-based preprocessing techniques to the context of LMUS extraction.

Definition 5 (L-resolvent). *The* L-resolvent *of two labelled clauses* $(x \vee A)^{L_1}$ *and* $(\bar{x} \vee B)^{L_2}$ *on variable* x *is the labelled clause* $(A \vee B)^{L_1 \cup L_2}$.

We will re-use the symbol $\otimes_x$ to denote the operation of L-resolution. As with the case of (plain) clauses, L-resolution rule is extended to sets of labelled clauses: for two such sets S_x and $S_{\bar{x}}$ of L-clauses which all contain the literal x and $\bar{x}$, resp., let $S_x \otimes_x S_{\bar{x}} = \{C_1^{L_1} \otimes_x C_2^{L_2} \mid C_1^{L_1} \in S_x, C_2^{L_2} \in S_{\bar{x}},$ and $C_1 \otimes_x C_2$ is not a tautology$\}$.

Labelled Variable Elimination. Given an LCNF formula Φ, with subformulas $\Phi_x = \{C^L \in \Phi \mid x \in C\}$ and $\Phi_{\bar{x}} = \{C^L \in \Phi \mid \bar{x} \in C\}$, similarly to the case of (plain) CNF, we define the operation $\mathsf{LVE}(\Phi, x) = (\Phi \setminus (\Phi_x \cup \Phi_{\bar{x}})) \cup (\Phi_x \otimes_x \Phi_{\bar{x}})$. Notice that (as with VE) the definition implies that for any $C^L \in \mathsf{LVE}(\Phi, x)$, we have $x \notin C$, and either (i) $C^L \in \Phi$, or (ii) there exist $(x \vee C_1)^{L_1}$ and $(\bar{x} \vee C_2)^{L_2}$ in Φ such that $C^L = (C_1 \vee C_2)^{L_1 \cup L_2}$, or both (i) and (ii). It is not difficult to see that $Cls(\mathsf{LVE}(\Phi, x)) = \mathsf{VE}(Cls(\Phi), x)$, that is, the set of (plain) clauses underlying the LCNF Φ undergoes the same transformation as it would without labels, modulo the repeated clauses. Hence, as with the case of VE, LVE preserves satisfiability.

We will now show that the presence of labels attached to the clauses during the variable elimination allows to keep track of the relationship between the pre- and post-elimination formulas, and, as a result, allows to perform elimination correctly, that is, any LMUS of $\mathsf{LVE}(\Phi, x)$ is also an LMUS of Φ. As a first step, we show that the operations of LVE and $|_M$ commute.

Lemma 1. *For any LCNF Φ, variable x, and set of labels M,* $\mathsf{LVE}(\Phi, x)|_M = \mathsf{LVE}(\Phi|_M, x)$.

Proof. Take any $C^L \in \mathsf{LVE}(\Phi, x)|_M$. Note that $L \subseteq M$ and $x \notin C$. By the definition of LVE, we have that either (i) $C^L \in \Phi$, or (ii) for some $(x \vee C_1)^{L_1}$ and $(\bar{x} \vee C_2)^{L_2}$ in Φ, we have $C = C_1 \vee C_2$ and $L_1 \cup L_2 = L$, or both (i) and (ii). In the case (i), the clause C^L is in $\Phi|_M$, since $L \subseteq M$, and since $x \notin C$, $C^L \in \mathsf{LVE}(\Phi|_M, x)$. In the case (ii), both clauses $(x \vee C_1)^{L_1}$ and $(\bar{x} \vee C_2)^{L_2}$ are in $\Phi|_M$, since $L_1 \cup L_2 = L \subseteq M$, and by the definition of LVE, $C^L = (C_1 \vee C_2)^{L_1 \cup L_2} \in \mathsf{LVE}(\Phi|_M, x)$.

For the opposite direction, take $C^L \in \mathsf{LVE}(\Phi|_M, x)$. Note that $x \notin C$. By the definition of LVE, we have that either (i) $C^L \in \Phi|_M$, or (ii) for some $(x \vee C_1)^{L_1}$ and $(\bar{x} \vee C_2)^{L_2}$ in $\Phi|_M$, we have $C = C_1 \vee C_2$ and $L_1 \cup L_2 = L$, or both (i) and (ii). In the case (i), since $C^L \in \Phi$ and $x \notin C$, by the definition of LVE, we have $C^L \in \mathsf{LVE}(\Phi, x)$, and since $C^L \in \Phi|_M$ we have $L \subseteq M$, and so $C^L \in \mathsf{LVE}(\Phi, x)|_M$. In the case (ii), since $(x \vee C_1)^{L_1}$ and $(\bar{x} \vee C_2)^{L_2}$ are in Φ, by the definition of LVE we have $C^L = (C_1 \vee C_2)^{L_1 \cup L_2} \in \mathsf{LVE}(\Phi, x)$; since both clauses are in $\Phi|_M$, we have $L_1 \subseteq M$ and $L_2 \subseteq M$, Hence $L = L_1 \cup L_2 \subseteq M$, and $C^L \in \mathsf{LVE}(\Phi, x)$. □

Correctness of LVE with respect to LMUS extraction is established by applying Lemma 1.

Theorem 1. *For any LCNF formula Φ and variable x, any LMUS of* $\mathsf{LVE}(\Phi, x)$ *is an LMUS of Φ.*

Proof. Let M be an LMUS of $\mathsf{LVE}(\Phi, x)$, i.e. $\mathsf{LVE}(\Phi, x)|_M \in \mathsf{UNSAT}$, and for any $M' \subset M$, $\mathsf{LVE}(\Phi, x)|_{M'} \in \mathsf{SAT}$. By Lemma 1, we have $\mathsf{LVE}(\Phi, x)|_M = \mathsf{LVE}(\Phi|_M, x)$, and so $\mathsf{LVE}(\Phi|_M, x) \in \mathsf{UNSAT}$, and since LVE preserves satisfiability, $\Phi|_M \in \mathsf{UNSAT}$. Similarly, for the $M' \subset M$, by Lemma 1, we have $\mathsf{LVE}(\Phi, x)|_{M'} = \mathsf{LVE}(\Phi|_{M'}, x)$, and so $\mathsf{LVE}(\Phi|_{M'}, x) \in \mathsf{SAT}$, and so $\Phi|_{M'} \in \mathsf{SAT}$. Hence, $\Phi|_M \in \mathsf{UNSAT}$ and for any $M' \subset M$, $\Phi|_{M'} \in \mathsf{SAT}$, that is, M is an LMUS of Φ. □

Notice that the presence of labels addresses the problems with resolution-based preprocessing techniques in plain and group MUS settings outlined in Section 3. For example, labels provide a way to represent the multiple group identity of resolvents: a resolvent of two clauses from different groups simply inherits the identity of both groups. Furthermore, in the context of plain MUS extraction, if a clause C can be obtained by resolving two pairs of clauses C_1, C_2 and C_3, C_4, then in the LCNF setting, we will have *two* L-clauses C^{L_1} and C^{L_2} with $L_1 \neq L_2$. Although this might impede the effectiveness of VE, the correctness with respect to MUS computation is guaranteed. In fact, in Section 6, we demonstrate empirically that in the context of group MUS extraction, the technique is still effective.

Labelled Unit Propagation. Notice that $\mathsf{BCP}(F, l)$ can be seen as the combination of (i) elimination of all clauses $(l \vee C_1)$, $\ldots$, $(l \vee C_k)$ that are subsumed by (l) in F, and (ii) $\mathsf{VE}(F', l)$, where F' is the CNF formula resulting from the subsumption elimination step w.r.t. (l). Hence, combining Proposition 6 and Theorem 1, we can define *labelled unit propagation* $\mathsf{LBCP}(\Phi, (l)^L)$ for a given LCNF Φ and labelled unit clause $(l)^L \in \Phi$ as the combination of (1) labelled subsumption with the restriction that for each $(l \vee C_i)_i^L \in \Phi$ we have $L \subseteq L_i$ (following Proposition 6), and (2) $\mathsf{LVE}(\Phi', l)$, where Φ' is the LCNF resulting from step (1). Therefore, in the group-MUS setting, BCP can be applied *within* any of the groups, and by propagating any of the unit clauses derived from group G_0 to the groups G_i for $i > 0$.

Labelled Self-subsuming Resolution. Recall that a step of self-subsuming resolution $\mathsf{SSR}(F, C, D, l) = (F \setminus D) \cup \{C \otimes_l D\}$ can be seen as first adding the resolvent $(C \otimes_l D)$ to F, and then applying subsumption elimination to remove the the clause $D \supset C \otimes_l D$. Hence we define *labelled self-subsuming resolution* $\mathsf{LSSR}(\Phi, C^{L_C}, D^{L_D}, l)$ as the combination of (1) computing the L–resolvent of C^{L_C} and D^{L_D}, and (2) applying labelled subsumption to remove D^{L_D} from the LCNF resulting from step (i), with the restriction that $L_C \cup L_D \subseteq L_D$, i.e., $L_C \subseteq L_D$. The correctness of LSSR is established in the following proposition.

Proposition 7. *Let Φ be any LCNF formula with two L-clauses C^{L_C} and D^{L_D} resolvable on the variable l and satisfying $L_C \subseteq L_D$. Then, any LMUS of the formula* $\mathsf{LSSR}(\Phi, C^{L_C}, D^{L_D}, l)$ *is an LMUS of Φ.*

Proof. Follows directly from the facts that the clauses $D^{L_D} \in \Phi$ and $C^{L_C} \otimes_l D^{L_D} \in \Phi'$ have the exact same set of labels L_D since $L_C \subseteq L_D$, and that $Cls(\Phi)$ and $Cls(\Phi')$ are logically equivalent. □

5.3 Applying the Labelled Preprocessing Techniques in Practice

The reduction from LMUS extraction to group MUS extraction and the subsequent discussion on the applicability of incremental SAT solvers to the LMUS computation problem (recall Section 4) suggest a simple way to implement most of the LCNF-based preprocessing techniques, namely LBCP, LVE, LSSR, and labelled subsumption elimination. As discussed before, given an LCNF formula Φ, add a fresh variable p_l for each $l \in Lbls(\Phi)$, and for every $C^L \in \Phi$ create a clause $(C \vee \bigvee_{l \in L} p_l)$. By F_Φ let us denote the resulting CNF formula (*not* the group-CNF discussed earlier). It is easy to see that the corresponding preprocessing techniques for plain CNF formulas can now be applied to F_Φ as long as VE is disallowed to eliminate the variables p_l. The resulting CNF formula F'_Φ is then mapped back into an LCNF formula Φ' by converting the variables p_l in the clauses into the label-sets of L-clauses to obtain the preprocessed version of Φ. The formula Φ' is then given to an LMUS computation algorithm. Based on the results presented in this section, the computed LMUS M of Φ' is an LMUS of Φ.

Connecting back to practical group MUS extraction, a simple way to apply the labelled preprocessing techniques within group MUS extraction is to exploit assumptions within an incremental SAT solver that incorporates the original non-labelled versions of the preprocessing techniques (recall the discussion on computing LMUSes in Section 4). We used this approach for the experiments described next.

6 Experimental Results

The aim of the experimental study was to evaluate the potential effectiveness of various preprocessing techniques in the context of group MUS extraction. The focus on group MUSes is due to the high relevance of the problem to a number of formal verification applications (e.g. model checking and equivalence checking). To this end, we integrated some of the preprocessing techniques discussed in this paper into the group MUS extractor MUSer2 [3]. Specifically, we implemented BCE, which, as shown in Section 3, can be applied to group CNF instances safely prior to group MUS extraction by simply disregarding the group identities of the clauses. To implement additional preprocessing techniques, we took advantage of the fact that MUSer2 is an *assumption-based* MUS extractor, and followed the recipe outlined in the previous section: we configured it to

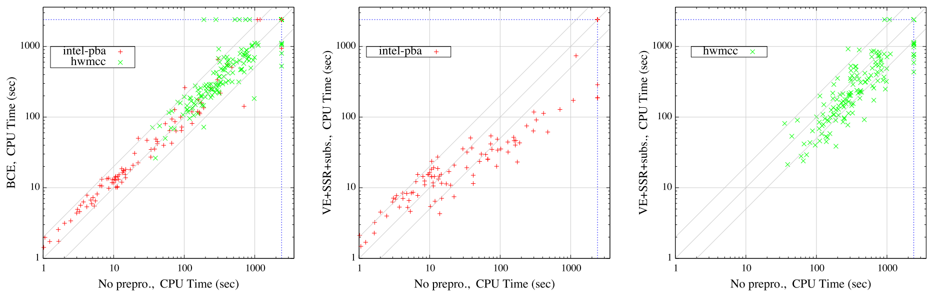


Fig. 6.1. Left: base vs. BCE. Center and right: base vs. VE+SSR+subsumption elimination (with SatElite), `intel-pba` center, `hwmcc` right. CPU time includes the time used for preprocessing.

work with the Minisat 2.2.0 (`http://minisat.se/`) SAT solver, and ran the SatElite preprocessor [6] of Minisat prior to group MUS extraction (note that SatElite allows to prohibit the elimination of particular variables). This corresponds to applying a combination of VE, SSR, and subsumption elimination prior to group MUS extraction

For the experiments, we used two sets of group MUS benchmarks. The first set, `intel-pba`, contains 99 instances submitted by Intel to the group MUS track of SAT Competition 2011. These instances originate from a proof-based abstraction framework. Their characteristic features are the size (reaching 4 million clauses), and the fact that over 90% of the clauses belong to group G_0. Each of the rest of the groups represents a gate (flop) over multiple timeframes in BMC unrolling. The second set, `hwmcc`, consists of 148 `belov` instances used in the same competition. These instances represent BMC unrolling of unsatisfiable instances from HWMCC 2010, whereby each AIG gate in each timeframe is represented a separate group (of 3 clauses). In these instances G_0 consists only of the unit clause that represent properly assertion. Note that hence the two sets differ drastically in structure, in a sense representing two extreme opposites in applications of group MUS extraction in proof-based abstraction.

The scatter plot on the left in Fig. 6.1, which demonstrates the effects of BCE on group MUS extraction time, suggests that BCE is not an effective technique for preprocessing group MUS instances. This is despite the fact that on most benchmarks BCE removes significant number of clauses (e.g. 2.5 million out of 3 on some of instances). On the other hand, as seen from the center and right plots in Fig. 6.1, the positive impact of resolution- and subsumption- based preprocessing on group MUS extraction time can be very significant, particularly on the difficult instances from the `intel-pba` set, where an order of magnitude speed-ups can be observed in some cases.

7 Conclusions

In this paper, we show that many CNF-level preprocessing techniques, routinely applied for speeding up SAT solving, are problematic in the context of plain MUS extraction, and, especially so, in the practically relevant context of group MUS extraction. To alleviate this problem, we developed sound liftings of the preprocessing techniques to the general context of labelled MUS extraction that captures group MUS extraction as well as various other forms of MUS extraction problems. Our experimental results show that label-based preprocessing can improve the efficiency of group MUS extraction.

References

1. Bacchus, F.: Enhancing Davis Putnam with extended binary clause reasoning. In: Proc. AAAI, pp. 613–619. AAAI Press (2002)
2. Belov, A., Marques-Silva, J.: Generalizing redundancy in propositional logic: Foundations and hitting sets duality. Tech. rep., arXiv (2012), http://arxiv.org/abs/1207.1257
3. Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor. J. SAT 8, 123–128 (2012)
4. Belov, A., Ivrii, A., Matsliah, A., Marques-Silva, J.: On Efficient Computation of Variable MUSes. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 298–311. Springer, Heidelberg (2012)
5. Desrosiers, C., Galinier, P., Hertz, A., Paroz, S.: Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. J. Comb. Optim. 18(2), 124–150 (2009)
6. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
7. Fourdrinoy, O., Grégoire, É., Mazure, B., Saïs, L.: Eliminating Redundant Clauses in SAT Instances. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 71–83. Springer, Heidelberg (2007)
8. Gershman, R., Strichman, O.: Cost-Effective Hyper-Resolution for Preprocessing CNF Formulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 423–429. Springer, Heidelberg (2005)
9. Grégoire, É., Mazure, B., Piette, C.: On approaches to explaining infeasibility of sets of Boolean clauses. In: Proc. ICTAI, pp. 74–83. IEEE (2008)
10. Han, H., Somenzi, F.: Alembic: An efficient algorithm for CNF preprocessing. In: Proc. DAC, pp. 582–587. IEEE (2007)
11. Heule, M.J.H., Järvisalo, M., Biere, A.: Efficient CNF Simplification Based on Binary Implication Graphs. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 201–215. Springer, Heidelberg (2011)
12. Heule, M., Järvisalo, M., Biere, A.: Clause Elimination Procedures for CNF Formulas. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 357–371. Springer, Heidelberg (2010)
13. Heule, M., Järvisalo, M., Biere, A.: Covered clause elimination. In: LPAR Short Paper (2010)
14. Järvisalo, M., Biere, A., Heule, M.: Blocked Clause Elimination. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 129–144. Springer, Heidelberg (2010)
15. Järvisalo, M., Heule, M., Biere, A.: Inprocessing Rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 355–370. Springer, Heidelberg (2012)
16. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics 96-97, 149–176 (1999)
17. Li, C., Manyà, F., Mohamedou, N., Planes, J.: Resolution-based lower bounds in MaxSAT. Constraints 15, 456–484 (2010)
18. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning 40(1), 1–33 (2008)
19. van Maaren, H., Wieringa, S.: Finding Guaranteed MUSes Fast. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 291–304. Springer, Heidelberg (2008)

20. Marques-Silva, J.: Computing minimally unsatisfiable subformulas: State of the art and future directions. J. Mult-Valued Log. S. 19(1-3), 163–183 (2012)
21. Marques-Silva, J., Lynce, I.: On Improving MUS Extraction Algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 159–173. Springer, Heidelberg (2011)
22. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Proc. FMCAD, pp. 221–229. IEEE (2010)
23. Ryvchin, V., Strichman, O.: Faster Extraction of High-Level Minimal Unsatisfiable Cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011)

Proof Tree Preserving Interpolation*

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz

Chair of Software Engineering, University of Freiburg

Abstract. Craig interpolation in SMT is difficult because, e. g., theory combination and integer cuts introduce mixed literals, i. e., literals containing local symbols from both input formulae. In this paper, we present a scheme to compute Craig interpolants in the presence of mixed literals. Contrary to existing approaches, this scheme neither limits the inferences done by the SMT solver, nor does it transform the proof tree before extracting interpolants. Our scheme works for the combination of uninterpreted functions and linear arithmetic but is extendable to other theories. The scheme is implemented in the interpolating SMT solver SMTInterpol.

1 Introduction

A Craig interpolant for a pair of formulae A and B whose conjunction is unsatisfiable is a formula I that follows from A and whose conjunction with B is unsatisfiable. Furthermore, I only contains symbols common to A and B. Model checking and state space abstraction [13,15] make intensive use of interpolation to achieve a higher degree of automation. This increase in automation stems from the ability to fully automatically generate interpolants from proofs produced by modern theorem provers.

For propositional logic, a SAT solver typically produces resolution-based proofs that show the unsatisfiability of an error path. Extracting Craig interpolants from such proofs is a well understood and easy task that can be accomplished, e. g., using the algorithms of Pudlák [19] or McMillan [14]. An essential property of the proofs generated by SAT solvers is that every proof step only involves literals that occur in the input.

This property does not hold for proofs produced by SMT solvers for formulae in a combination of first order theories. Such solvers produce new literals for different reasons. First, to combine two theory solvers, SMT solvers exchange (dis-)equalities between the symbols common to these two theories in a Nelson-Oppen-style theory combination. Second, various techniques dynamically generate new literals to simplify proof generation. Third, new literals are introduced in the context of a branch-and-bound or branch-and-cut search for non-convex theories. The theory of linear integer arithmetic for example is typically solved by

* This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR14 AVACS).

searching a model for the relaxation of the formula to linear rational arithmetic and then using branch-and-cut with Gomory cuts or *extended branches* [7] to remove the current non-integer solution from the solution space of the relaxation.

The literals produced by either of these techniques only contain symbols that are already present in the input. However, a literal produced by one of these techniques may be *mixed*[1] in the sense that it may contain symbols occurring only in A and symbols occurring only in B. These literals pose the major difficulty when extracting interpolants from proofs produced by SMT solvers.

In this paper, we present a scheme to compute Craig interpolants in the presence of mixed literals. Our interpolation scheme is based on syntactical restrictions of *partial interpolants* and specialized rules to interpolate resolution steps on mixed literals. This enables us to compute interpolants in the context of a state-of-the-art SMT solver without manipulating the proof tree or restricting the solver in any way. We base our presentation on the quantifier-free fragment of the combined theory of uninterpreted functions and linear arithmetic over the rationals or the integers. The interpolation scheme is used in the interpolating SMT solver SMTInterpol [4]. Proofs for the theorems in this paper are given in the technical report [3].

Related Work. For Boolean circuits, Pudlák [19] shows how to construct quantifier-free interpolants from resolution proofs of unsatisfiability. A different proof-based interpolation system is given by McMillan [14] in his seminal paper on interpolation for SMT. The presented method combines the theory of equality and uninterpreted functions with the theory of linear rational arithmetic. Interpolants are computed from partial interpolants by annotating every proof step. The partial interpolants have a specific form that carries information needed to combine the theories. The proof system is incomplete for linear integer arithmetic as it cannot deal with arbitrary cuts and mixed literals introduced by these cuts.

Brillout et al. [1] present an interpolating sequent calculus that can compute interpolants for the combination of uninterpreted functions and linear integer arithmetic. The interpolants computed using their method might contain quantifier since they do not use divisibility predicates. Furthermore their method limits the generation of Gomory cuts in the integer solver to prevent some mixed cuts. The method presented in this paper combines the two theories without quantifiers and, furthermore, does not restrict any component of the solver.

Yorsh and Musuvathi [20] show how to combine interpolants generated by an SMT solver based on Nelson-Oppen combination. They define the concept of *equality-interpolating theories*. These are theories that can provide a shared term t for a mixed literal $a = b$ that is derivable from an interpolation problem. A troublesome mixed interface equality $a = b$ is rewritten into the conjunction $a = t \wedge t = b$. They show that both, the theory of uninterpreted functions and the theory of linear rational arithmetic are equality-interpolating. We do not

[1] Mixed literals sometimes are called *uncolorable*.

explicitly split the proof. Additionally, our method can handle the theory of linear integer arithmetic without any restriction on the solver.

Cimatti et al. [5] present a method to compute interpolants for linear rational arithmetic and difference logic. The method presented in this paper builds upon their interpolation technique for linear rational arithmetic. For theories combined via delayed theory combination, they show how to compute interpolants by transforming a proof into a so-called *ie-local* proof. In these proofs, mixed equalities are close to the leaves of the proof tree and splitting them is cheap since the proof trees that have to be duplicated are small.

Goel et al. [11] present a generalization of equality-interpolating theories. They define the class of *almost-colorable proofs* and an algorithm to generate interpolants from such proofs. Furthermore they describe a restricted DPLL system to generate almost-colorable proofs. This system does not restrict the search if convex theories are used. Their procedure is incomplete for non-convex theories like linear arithmetic over integers since it prohibits the generation of mixed branches and cuts.

Recently, techniques to transform proofs gained a lot of attention. Bruttomesso et al. [2] present a framework to lift resolution steps on mixed literals into the leaves of the resolution tree. Once a subproof only resolves on mixed literals, they replace this subproof with the conclusion removing the mixed inferences. The newly generated lemmas however are mixed between different theories and require special interpolation procedures. Even though these procedures only have to deal with conjunctions of literals in the combined theories it is not obvious how to compute interpolants in this setting. In contrast to our approach, they manipulate the proof in a way that is worst-case exponential and rely on an interpolant generator for the conjunctive fragment of the combined theories.

McMillan [16] presents a technique to compute interpolants from Z3 proofs. Whenever a sub-proof contains mixed literals, he extracts lemmas from the proof tree and delegates them to a second (possibly slower) interpolating solver.

2 Preliminaries

Logic, Theories, and SMT. We assume standard first-order logic. We operate within the quantifier-free fragments of the theory of equality with uninterpreted functions $\mathcal{EUF}$ and the theories of linear arithmetic on rationals $\mathcal{LA}(\mathbb{Q})$ and integers $\mathcal{LA}(\mathbb{Z})$. The quantifier-free fragment of $\mathcal{LA}(\mathbb{Z})$ is not closed under interpolation. Therefore, we augment the signature with division by constant functions $\lfloor \frac{\cdot}{k} \rfloor$ for all integers $k \geq 1$.

We use the standard notations $\models_T, \bot, \top$ to denote entailment in the theory T, contradiction, and tautology. In the following, we drop the subscript T as it always corresponds to the combined theory of $\mathcal{EUF}$, $\mathcal{LA}(\mathbb{Q})$, and $\mathcal{LA}(\mathbb{Z})$.

The literals in $\mathcal{LA}(\mathbb{Z})$ are of the form $s \leq c$, where c is an integer constant and s a linear combination of variables. For $\mathcal{LA}(\mathbb{Q})$ we use constants $c \in \mathbb{Q}_\varepsilon$, $\mathbb{Q}_\varepsilon := \mathbb{Q} \cup \{q - \varepsilon | q \in \mathbb{Q}\}$ where the meaning of $s \leq q - \varepsilon$ is $s < q$. For better readability we use, e. g., $x \leq y$ resp. $x > y$ to denote $x - y \leq 0$ resp. $y - x \leq -\varepsilon$. In the integer case we use $x > y$ to denote $y - x \leq -1$.

Our algorithm operates on a proof of unsatisfiability generated by an SMT solver based on DPLL(T) [18]. Such a proof is a resolution tree with the $\bot$-clause at its root. The leaves of the tree are either clauses from the input formulae[2] or theory lemmas that are produced by one of the theory solvers. The negation of a theory lemma is called a *conflict*.

The theory solvers for $\mathcal{EUF}$, $\mathcal{LA}(\mathbb{Q})$, and $\mathcal{LA}(\mathbb{Z})$ are working independently and exchange (dis-)equality literals through the DPLL engine in a Nelson-Oppen style [17]. Internally, the solver for linear arithmetic uses only inequalities in theory conflicts. In the proof tree, the (dis-)equalities are related to inequalities by the (valid) clauses $x = y \vee x < y \vee x > y$, and $x \neq y \vee x \leq y$. We call these leaves of the proof tree *theory combination clauses.*

Interpolants and Symbol Sets. For a formula F, we use $symb(F)$ to denote the set of non-theory symbols occurring in F. An interpolation problem is given by two formulae A and B such that $A \wedge B \models \bot$. An interpolant of A and B is a formula I such that (i) $A \models I$, (ii) $B \wedge I \models \bot$, and (iii) $symb(I) \subseteq symb(A) \cap symb(B)$.

We call a symbol $s \in symb(A) \cup symb(B)$ *shared* if $s \in symb(A) \cap symb(B)$, *A-local* if $s \in symb(A) \setminus symb(B)$, and *B-local* if $s \in symb(B) \setminus symb(A)$. Similarly, we call a term *A-local* (*B-local*) if it contains at least one A-local (B-local) and no B-local (A-local) symbols. We call a term *(AB-)shared* if it contains only shared symbols and *(AB-)mixed* if it contains A-local as well as B-local symbols. The same terminology applies to formulae.

Substitution in Formulae and Monotonicity. By $F[G]$ we denote a formula in negation normal form with a sub-formula G that occurs positively in the formula. Substituting this sub-formula by a formula G' is denoted by $F[G']$. By $F(t)$ we denote a formula with a sub-term t that can appear anywhere in F. The substitution of t with a term t' is denoted by $F(t')$.

The following lemma is important for the correctness proofs of our interpolation scheme.

Lemma 1 (Monotonicity). *Given a formula $F[G_1][G_2]\dots[G_n]$ in negation normal form with sub-formulae $G_1, G_2, \dots, G_n$ occurring only positively in the formula and formulae $G'_1, \dots, G'_n$, it holds that*

$$\bigwedge_{i \in \{1,\dots,n\}} (G_i \rightarrow G'_i) \rightarrow (F[G_1]\dots[G_n] \rightarrow F[G'_1]\dots[G'_n])$$

3 Proof Tree-Based Interpolation

Interpolants can be computed from proofs of unsatisfiability as Pudlák and McMillan have already shown. In this section we will introduce their algorithms. Then, we will discuss the changes necessary to handle mixed literals introduced, e. g., by theory combination.

2 W. l. o. g. we assume input formulae are in conjunctive normal form.

3.1 Pudlák's and McMillan's Interpolation Algorithms

Pudlák's and McMillan's algorithms assume that the literals are not mixed. We will remove this restriction later. We define a common framework that is more general and can be instantiated to obtain Pudlák's or McMillan's algorithm to compute interpolants. For this, we use two projection functions on literals $\cdot \downharpoonleft A$ and $\cdot \downharpoonleft B$ as defined below. They have the properties (i) $symb(\ell \downharpoonleft A) \subseteq symb(A)$, (ii) $symb(\ell \downharpoonleft B) \subseteq symb(B)$, and (iii) $\ell \iff (\ell \downharpoonleft A \wedge \ell \downharpoonleft B)$. Other projection functions are possible and this allows for varying the strength of the resulting interpolant as shown in [8]. We extend the projection function to conjunctions of literals component-wise.

	Pudlák		McMillan	
	$\ell \downharpoonleft A$	$\ell \downharpoonleft B$	$\ell \downharpoonleft A$	$\ell \downharpoonleft B$
ℓ is A-local	ℓ	$\top$	ℓ	$\top$
ℓ is B-local	$\top$	ℓ	$\top$	ℓ
ℓ is shared	ℓ	ℓ	$\top$	ℓ

Given an interpolation problem A and B, a *partial interpolant* of a clause C is an interpolant of the formulae $A \wedge (\neg C \downharpoonleft A)$ and $B \wedge (\neg C \downharpoonleft B)$[3]. Partial interpolants can be computed inductively over the structure of the proof tree. A partial interpolant of a theory lemma C can be computed by a theory-specific interpolation routine as an interpolant of $\neg C \downharpoonleft A$ and $\neg C \downharpoonleft B$. Note that the conjunction is equivalent to $\neg C$ and therefore unsatisfiable. For an input clause C from the formula A (resp. B), a partial interpolant is $\neg(\neg C \setminus A)$ (resp. $\neg C \setminus B$) where $\neg C \setminus A$ is the conjunction of all literals of $\neg C$ that are not in $\neg C \downharpoonleft A$ and analogously for $\neg C \setminus B$. For a resolution step, a partial interpolant can be computed using (rule-res), which is given below. For this rule, it is easy to show that I_3 is a partial interpolant of $C_1 \vee C_2$ given that I_1 and I_2 are partial interpolants of $C_1 \vee \ell$ and $C_2 \vee \neg\ell$, respectively. Note that the "otherwise" case never triggers in McMillan's algorithm.

$$\frac{C_1 \vee \ell : I_1 \quad C_2 \vee \neg\ell : I_2}{C_1 \vee C_2 : I_3} \quad \text{where } I_3 = \begin{cases} I_1 \vee I_2 & \text{if } \ell \downharpoonleft B = \top \\ I_1 \wedge I_2 & \text{if } \ell \downharpoonleft A = \top \\ (I_1 \vee \ell) \wedge (I_2 \vee \neg\ell) & \text{otherwise} \end{cases} \qquad \text{(rule-res)}$$

As the partial interpolant of the root of the proof tree (which is labelled with the clause $\bot$) is an interpolant of the input formulae A and B, this algorithm can be used to compute interpolants.

Theorem 1. *The above-given partial interpolants are correct, i.e., if I_1 is a partial interpolant of $C_1 \vee \ell$ and I_2 is a partial interpolant of $C_2 \vee \neg\ell$ then I_3 is a partial interpolant of the clause $C_1 \vee C_2$.*

[3] Note that $\neg C$ is a conjunction of literals. Thus, $\neg C \downharpoonleft A$ is well defined.

3.2 Purification of Mixed Literals

The proofs generated by state-of-the-art SMT solvers may contain mixed literals. We tackle them by extending the projection functions to these literals. The problem here is that there is no projection function that satisfies the conditions in the previous section. Therefore, we relax the conditions by allowing fresh auxiliary variables to occur in the projections.

In our setting there are two different kinds of mixed literals: First, (dis-)equalities of the form $a = b$ or $a \neq b$ for an A-local variable a and a B-local variable b are introduced, e. g., by theory combination or Ackermannization. Second, inequalities of the form $a + b \leq c$ are introduced, e. g., by extended branches [7] or bound propagation. Here, a is a linear combination of A-local variables, b is a linear combination of B-local and shared variables, and c is a constant.

We split mixed literals using auxiliary variables, which we denote by x, x_a, or x_b in the following. One or two fresh variables are introduced for each mixed literal. We count these variables as shared between A and B. The purpose of the auxiliary variables is to capture the shared value that needs to be propagated between A and B. When splitting a literal ℓ into A- and B-part, we require that $\ell \Leftrightarrow \exists x, x_a, x_b.(\ell \downharpoonleft A) \wedge (\ell \downharpoonleft B)$. We need two variables x_a and x_b to split the literal $a \neq b$ into two symmetric parts. For symmetry we split the literal $a = b$ in the same fashion instead of introducing only a single auxiliary variable. This is achieved by the definitions below.

$$\begin{aligned}
(a = b) \downharpoonleft A &:= (a = x_a \wedge x_a = x_b) & (a = b) \downharpoonleft B &:= (x_a = x_b \wedge x_b = b)\\
(a \neq b) \downharpoonleft A &:= (a = x_a \wedge x_a \neq x_b) & (a \neq b) \downharpoonleft B &:= (x_a \neq x_b \wedge x_b = b)\\
(a + b \leq c) \downharpoonleft A &:= (a + x \leq 0) & (a + b \leq c) \downharpoonleft B &:= (-x + b \leq c)
\end{aligned}$$

Since the mixed variables are considered to be shared, we allow them to occur in the partial interpolant of a clause C. However, a variable may only occur if C contains the corresponding literal. This is achieved by a special interpolation rule for resolution steps where the pivot literal is mixed. The rules for the different mixed literals are the core of our proposed algorithm and will be introduced in the following sections.

Instead of with a single partial interpolant, we label each clause with a pattern from which we can derive two partial interpolants, a strong and a weak one. The strong interpolant of a clause C implies the weak interpolant under the assumption that $\neg C \downharpoonleft A$ or $\neg C \downharpoonleft B$ holds. Having two interpolants enables us to complete the inductive proof. We show that the strong interpolant follows from the A-part of the resolvent if the strong interpolants of the premises follow from their respective A-part. On the other hand, the weak interpolant is in contradiction to the B-part in the resolvent if this is the case for the premises. Since the weak interpolant follows from the strong interpolant this shows that both are partial interpolants. The models for the strong and the weak interpolants only differ in the values of the auxiliary variable. The interpolants are needed because the "right" value for the auxiliary variable is not known when interpolating the leaves of the proof tree. The strong and the weak interpolant are identical if the clause does not contain mixed literals. Therefore, we derive only one interpolant for the bottom clause.

It is important to state here that the given purification of a literal into two new literals is not a modification of the proof tree or any of its nodes. The proof tree would no longer be well-formed if we replaced a mixed literal by the disjunction or conjunction of the purified parts. The purification is only used to define partial interpolants of clauses. In fact, it is only used in the correctness proof of our method and is not even done explicitly in the implementation.

4 Uninterpreted Functions

In this section we will present the part of our algorithm that is specific to the theory $\mathcal{EUF}$. The only mixed atom that is considered by this theory is $a = b$ where a is A-local and b is B-local.

4.1 Leaf Interpolation

The $\mathcal{EUF}$ solver is based on the congruence closure algorithm [6]. The theory lemmas are generated from conflicts involving a single disequality that is in contradiction to a path of equalities. Thus, the clause generated from such a conflict consists of a single equality literal and several disequality literals.

When computing the partial interpolants of the theory lemmas, we internally split the mixed literals according to Section 3.2. Then we use an algorithm similar to [10] to compute an interpolant. This algorithm basically summarises the A-equalities that are adjacent on the path of equalities.

If the theory lemma contains a mixed equality $a = b$ (without negation), it corresponds to the single disequality in the conflict. The disequality is split into $a = x_a$, $x_a \neq x_b$ and $x_b = b$ and the resulting interpolant depends on whether we consider the disequality to belong to the A-part or to the B-part. If we consider it to belong to the B-part, then x_a is the end of an equality path summing up the equalities from A. Thus, the computed interpolant has the form $I[x_a = s]$. If we consider $x_a \neq x_b$ to belong to the A-part, the resulting interpolant is $I[x_b \neq s]$. Note that in both cases the literal $x_a = s$ resp. $x_b \neq s$ occurs positively in the interpolant and is the only literal containing x_a resp. x_b. To summarise, the partial interpolant computed for a theory clause $C \vee a = b$ where $a = b$ has the auxiliary variables x_a, x_b has the form $I[x_a = s]$ or $I[x_b \neq s]$ and x_a, x_b do not appear at any other place in I. Both interpolants $I[x_a = s]$ and $I[x_b \neq s]$ are partial interpolants of the clause. From $x_a \neq x_b$ we can derive the weak interpolant $I[x_b \neq s]$ from the strong interpolant $I[x_a = s]$ using Lemma 1 (monotonicity). We define

$$EQ_S(x, s) := (x_a = s), \qquad EQ_W(x, s) := (x_b \neq s)$$

and label a clause in the proof tree with $I[EQ(x,s)]$ to denote that the formulae $I[EQ_S(x,s)]$ and $I[EQ_W(x,s)]$ are the strong and weak partial interpolants.

For theory lemmas containing the literal $a \neq b$, the corresponding auxiliary variables x_a, x_b may appear anywhere in the partial interpolant, even under a function symbol. A simple example is the theory conflict $s \neq f(a) \land a = (x_a = x_b =) b \land f(b) = s$, which has the partial interpolants $s \neq f(x_a)$ and $s \neq f(x_b)$ (depending on whether $x_a = x_b$ is considered as A- or as B-literal). We simply label the corresponding theory lemma with the interpolant $s \neq f(x)$. In general the label of such a clause has the form $I(x)$. The formulae $I(x_a)$ and $I(x_b)$ are the strong and weak partial interpolants of that clause. Of course, here the interpolants are equivalent given $x_a = x_b$.

When two partial interpolants for clauses containing $a = b$ are combined using (rule-res), i. e., the pivot literal is a non-mixed literal but the mixed literal $a = b$ occurs in C_1 and C_2, the resulting partial interpolant may contain $EQ(x,s_1)$ and $EQ(x,s_2)$ for different shared terms s_1, s_2. In general, we allow the partial interpolants to have the form $I[EQ(x,s_1)]\dots[EQ(x,s_n)]$.

4.2 Pivoting of Mixed Equalities

We require that every clause containing $a = b$ with auxiliary variables x_a, x_b is always labelled with a formula of the form $I[EQ(x,s_1)]\dots[EQ(x,s_n)]$ and that this is a partial interpolant of the clause for both EQ_S and EQ_W. As discussed above, this is automatically the case for the theory lemmas computed from conflicts in the congruence closure algorithm. This property is also preserved by (rule-res) and this rule also preserves the property of being a strong or weak partial interpolant.

On the other hand, a clause containing the literal $a \neq b$ is labelled with a formula of the form $I(x)$, i. e., the auxiliary variable x can occur at arbitrary positions. Both $I(x_a)$ and $I(x_b)$ are partial interpolants of the clause. Again, the form $I(x)$ and the property of being a partial interpolant is also preserved by (rule-res).

We use the following rule to interpolate the resolution step on the mixed literal $a = b$.

$$\frac{C_1 \lor a = b : I_1[EQ(x,s_1)]\dots[EQ(x,s_n)] \qquad C_2 \lor a \neq b : I_2(x)}{C_1 \lor C_2 : I_1[I_2(s_1)]\dots[I_2(s_n)]} \qquad \text{(rule-eq)}$$

The rule replaces every literal $EQ(x,s_i)$ in I_1 with the formula $I_2(s_i)$, in which every x is substituted by s_i. Therefore the auxiliary variable introduced for the mixed literal $a = b$ is removed.

Theorem 2 (Soundness of (rule-eq)). *Let $a = b$ be a mixed literal with auxiliary variable x. If $I_1[EQ(x,s_1)]\dots[EQ(x,s_n)]$ yields two (strong and weak) partial interpolants of $C_1 \lor a = b$ and $I_2(x)$ two partial interpolants of $C_1 \lor a \neq b$ then $I_1[I_2(s_1)]\dots[I_2(s_n)]$ yields two partial interpolants of the clause $C_1 \lor C_2$.*

4.3 Example

We demonstrate our algorithm on the following example:

$$A \equiv (\neg p \vee a = s_1) \wedge (p \vee a = s_2) \wedge f(a) = t$$
$$B \equiv (\neg p \vee b = s_1) \wedge (p \vee b = s_2) \wedge f(b) \neq t$$

The conjunction $A \wedge B$ is unsatisfiable. In this example, a is A-local, b is B-local and the remaining symbols are shared.

Assume the theory solver for $\mathcal{EUF}$ introduces the mixed literal $a = b$ and provides the lemmas (i) $f(a) \neq t \vee a \neq b \vee f(b) = t$, (ii) $a \neq s_1 \vee b \neq s_1 \vee a = b$, and (iii) $a \neq s_2 \vee b \neq s_2 \vee a = b$. Let the variable x be associated with the equality $a = b$. Then, we label the lemmas with (i) $f(x) = t$, (ii) $EQ(x, s_1)$, and (iii) $EQ(x, s_2)$.

We compute an interpolant for A and B using Pudlák's algorithm. Since the input is already in clausal form, we can directly apply resolution. From lemma (ii) and the input clauses $\neg p \vee a = s_1$ and $\neg p \vee b = s_1$ we can derive the clause $\neg p \vee a = b$. The partial interpolant of the derived clause is still $EQ(x, s_1)$, since the partial interpolants of the input clauses are $\perp$ resp. $\top$. Similarly, from lemma (iii) and the input clauses $p \vee a = s_2$ and $p \vee b = s_2$ we can derive the clause $p \vee a = b$ with partial interpolant $EQ(x, s_2)$. A resolution step on these two clauses with p as pivot yields the clause $a = b$. Since p is a shared literal, Pudlák's algorithm introduces the case distinction. Hence, we get the partial interpolant $(EQ(x, s_2) \vee p) \wedge (EQ(x, s_1) \vee \neg p)$. Note that this interpolant has the form $I_1[EQ(x, s_1)][EQ(x, s_2)]$ and, therefore, satisfies the syntactical restrictions.

From the $\mathcal{EUF}$-lemma (i) and the input clauses $f(a) = t$ and $f(b) \neq t$, we can derive the clause $a \neq b$ with partial interpolant $f(x) = t$. Note that this interpolant has the form $I_2(x)$ which also corresponds to the syntactical restrictions needed for our method.

If we apply the final resolution step on the mixed literal $a = b$ using (rule-eq), we get the interpolant $I_1[I_2(s_1)][I_2(s_2)]$ which corresponds to the interpolant $(f(s_2) = t \vee p) \wedge (f(s_1) = t \vee \neg p)$.

5 Linear Real and Integer Arithmetic

Our solver for linear arithmetic is based on a variant of the Simplex approach [9]. A theory conflict is a conjunction of literals ℓ_j of the form $\sum_i a_{ij} x_i \leq b_j$. The proof of unsatisfiability is given by Farkas coefficients $k_j \geq 0$ for each inequality ℓ_j. These coefficients have the properties $\sum_j k_j a_{ij} = 0$ and $\sum_j k_j b_j < 0$. In the following we use the notation of adding inequalities (provided the coefficients are positive). Thus, we write $\sum_j k_j \ell_j$ for $\sum_i (\sum_j k_j a_{ij}) x_i \leq \sum_j k_j b_j$. With the property of the Farkas coefficients we get a contradiction $(0 < 0)$ and this shows that the theory conflict is unsatisfiable.

A conjunction of literals may have rational but no integer solutions. In this case, there are no Farkas coefficients that can prove the unsatisfiability. So for the integer case, our solver may introduce an extended branch [7], which is just a branch of the DPLL engine on a newly introduced literal. In the proof tree this results in a resolution step with this literal as pivot.

Example 1. The formula $t \leq 2a \leq r \leq 2b + 1 \leq t$ has no integer solution but a rational solution. Introducing the branch $a \leq b \vee b < a$ leads to the theory conflicts $t \leq 2a \leq 2b \leq t - 1$ and $r \leq 2b + 1 \leq 2a - 1 \leq r - 1$ (note that $b < a$ is equivalent to $b + 1 \leq a$). The corresponding proof tree is given below. The Farkas coefficients in the theory lemmas are given in parenthesis. Note that the proof tree shows the clauses, i. e., the negated conflicts. A node with more than two parents denotes that multiple applications of the resolution rule are taken one after another.

$$\begin{tikzcd}
\begin{array}{r}\neg(r \leq 2b+1)\ (\cdot 1)\\ \neg(b+1 \leq a)\ (\cdot 2)\\ \neg(2a \leq r)\ (\cdot 1)\end{array} \arrow[dr, no head] & r \leq 2b+1 \arrow[d, no head] & 2a \leq r \arrow[dl, no head] & \begin{array}{r}\neg(t \leq 2a)\ (\cdot 1)\\ \neg(a \leq b)\ (\cdot 2)\\ \neg(2b+1 \leq t)\ (\cdot 1)\end{array} \arrow[dr, no head] & t \leq 2a \arrow[d, no head] & 2b+1 \leq t \arrow[dl, no head] \\
 & a \leq b \arrow[dr, no head] & & & \neg(a \leq b) \arrow[dll, no head] & \\
 & & \perp & & &
\end{tikzcd}$$

Now consider the problem of deriving an interpolant between $A \equiv t \leq 2a \leq r$ and $B \equiv r \leq 2b + 1 \leq t$. We can obtain an interpolant by annotating the above resolution tree with partial interpolants. Using the purification and summing up the contributions of the A-part we get the partial interpolants $2x_1 \leq r$ for $a \leq b$ and $2x_2 + t \leq 0$ for $\neg(a \leq b)$. Intuitively, the variable x_1 stands for a and x_2 for $-a$. Summing up the two partial interpolants with $x_1 = -x_2$ we get $t \leq r$. While this follows from A, it is not inconsistent with B. We need an additional argument that, given $r = t$, r has to be an even integer. This also follows from the A-part, more precisely from $t \leq -2x_2 = 2x_1 \leq r$. The final interpolant computed by our algorithm is $t \leq r \wedge (t \geq r \rightarrow t \leq 2\lfloor r/2 \rfloor)$.

In general, we can derive additional constraints on the variables if the constraint resulting from summing up the two partial interpolants holds very tightly. We know implicitly that $x_1 = -x_2$ is an integer value between $t/2$ and $r/2$. If t equals r or almost equals r there are only a few possible values which we can explicitly express using the division function as in the example above. This leads to the general form $t - r \leq 0 \wedge (t - r \geq -k \rightarrow F)$. In our example we have $k = 0$ and F specifies that $r = t$ is even.

To mechanise the reasoning used in the example above, our resolution rule for mixed inequality literals requires that the interpolant patterns that label the clauses have a certain shape. An auxiliary variable of a mixed inequality literal may only occur in the interpolant pattern if the negated literal appears in the clause. Let $\boldsymbol{x}$ denote the set of the variables that occur in the pattern. We additionally require that these variables only occur inside a special sub-formula of the form $LA(s(\boldsymbol{x}), k, F(\boldsymbol{x}))$. The first parameter s is a linear term over the variables in $\boldsymbol{x}$ and arbitrary other terms not involving $\boldsymbol{x}$. The coefficients of the variables $\boldsymbol{x}$ in s must all be positive. The second parameter $k \in \mathbb{Q}_\varepsilon$ is a constant value. In the real case we only allow the values 0 and $-\varepsilon$, in the integer case we allow $k \in \mathbb{Z}, k \geq -1$. The third parameter $F(\boldsymbol{x})$ is a formula that contains the variables from $\boldsymbol{x}$ at arbitrary positions. To simplify the presentation, we treat $-\varepsilon$ as -1 in the integer case. Again we have a strong and a weak partial interpolant that are obtained by using different definitions for LA. These definitions are

$$LA_S\,(s(\boldsymbol{x}), k, F(\boldsymbol{x})) :\equiv \forall \boldsymbol{x}' \leq \boldsymbol{x} : s(\boldsymbol{x}') \leq 0 \wedge (s(\boldsymbol{x}') \geq -k \rightarrow F(\boldsymbol{x}'))$$
$$LA_W\,(s(\boldsymbol{x}), k, F(\boldsymbol{x})) :\equiv \exists \boldsymbol{x}' \geq \boldsymbol{x} : s(\boldsymbol{x}') \leq 0 \wedge (s(\boldsymbol{x}') \geq -k \rightarrow F(\boldsymbol{x}'))$$

The intuition behind the formula $LA(s(\boldsymbol{x}), k, F(\boldsymbol{x}))$ is that $s(\boldsymbol{x}) \leq 0$ summarises the inequality chain that follows from the A-part of the formula. On this chain there may be some constraints on intermediate values. In the example above the A-part contains the chain $t \leq 2a \leq r$, which is summarised to $t \leq r$. Furthermore the A-part implies that there is an even integer value between t and r. If t and r are distinct, this is no problem. However, if $t \geq r$ we need that t is even. Using the above pattern we can choose $k = 0$ and F as the formula that states that t is even.

To see that the strong interpolant $LA_S(s, k, F)$ implies the weak interpolant $LA_W(s, k, F)$, instantiate $\boldsymbol{x}'$ with $\boldsymbol{x}$ in both formulas. Having quantifiers in the interpolant is no problem; once all mixed literals are resolved, all auxiliary variables are removed. Then, the strong and weak interpolant are identical and have no quantifiers.

In the remainder of the section, we will give the interpolants for the leaves produced by the linear arithmetic solver and for the resolvent of the resolution step where the pivot is a mixed linear inequality.

5.1 Leaf Interpolation

As mentioned above, our solver produces for a clause $C = \neg\ell_1 \vee \cdots \vee \neg\ell_m$ some Farkas coefficients $k_1, \ldots, k_m \geq 0$ such that $\sum_j k_j \ell_j$ yields a contradiction $0 < 0$. The interpolant for a theory lemma can be computed by summing up the A-part of the conflict: I is defined as $\sum_j k_j(\ell_j \downharpoonright A)$ (if $\ell_j \downharpoonright A = \top$ we regard it as $0 \leq 0$, i. e., it is not added to the sum). It is a valid interpolant as it clearly follows from $\neg C \downharpoonright A \iff \ell_1 \downharpoonright A \wedge \cdots \wedge \ell_m \downharpoonright A$. Moreover, we have that $I + \sum_j k_j(\ell_j \downharpoonright B)$ yields $0 < 0$, since for every literal, even for mixed literals, $\ell_j \downharpoonright A + \ell_j \downharpoonright B = \ell_j$ holds. This shows that $I \wedge \neg C \downharpoonright B$ is unsatisfiable.

The linear constraint $\sum_j k_j(\ell_j \downharpoonright A)$ can easily be expressed as $s(\boldsymbol{x}) \leq 0$. Thus, we can equivalently write the interpolant in our pattern as $LA(s(\boldsymbol{x}), -\varepsilon, \bot)$. Since the Farkas coefficients are all positive and the auxiliary variables introduced to define $\ell \downharpoonright A$ for mixed literals contain x positively, the resulting term $s(\boldsymbol{x})$ will also always contain x with a positive coefficient.

Theory combination lemmas. As mentioned in the preliminaries, we use theory combination clauses to propagate equalities from and to the Simplex core of the linear arithmetic solver. These clauses must also be labelled with partial interpolants. Table 1 shows the corresponding partial interpolants. The non-mixed case is given in the technical report.

Table 1. Interpolation of mixed theory combination clauses. We assume a is A-local, b is B-local, $a - b \leq 0$ has the auxiliary variable x_1, $b - a \leq 0$ has the auxiliary variable x_2 and $a = b$ the auxiliary variables x_a and x_b.

Clause C: $a \neq b \vee a \leq b$	Clause C: $a \neq b \vee b \leq a$
$\neg C \downharpoonleft A$: $a = x_a \wedge x_a = x_b \wedge -a + x_1 \leq 0$	$\neg C \downharpoonleft A$: $a = x_a \wedge x_a = x_b \wedge a + x_2 \leq 0$
$\neg C \downharpoonleft B$: $x_a = x_b \wedge x_b = b \wedge -x_1 + b < 0$	$\neg C \downharpoonleft B$: $x_a = x_b \wedge x_b = b \wedge -x_2 - b < 0$
Interpolant I: $LA(-x + x_1, -\varepsilon, \bot)$	Interpolant I: $LA(x + x_2, -\varepsilon, \bot)$

Clause C: $a = b \vee a < b \vee a > b$
$\neg C \downharpoonleft A$: $a = x_a \wedge x_a \neq x_b \wedge -a + x_1 \leq 0 \wedge a + x_2 \leq 0$
$\neg C \downharpoonleft B$: $a = x_a \wedge x_a \neq x_b \wedge -x_1 + b \leq 0 \wedge -x_2 - b \leq 0$
Interpolant I: $LA(x_1 + x_2, 0, EQ(x, x_1))$

The interpolant for the clause $a = b \vee a < b \vee a > b$ deserves more explanation. This clause is used to propagate equalities from the linear arithmetic solver if it can derive $a \leq b$ and $b \leq a$. In the interpolant, x_1 is the variable with $b \leq x_1 \leq a$, and x_2 the variable with $a \leq -x_2 \leq b$. The formula $LA(x_1 + x_2, 0, EQ(x, x_1))$ basically states that $x_1 \leq -x_2$ and that if $x_1 \geq -x_2$ then x_1 equals the shared value x of the equality $a = b$. We stress that the interpolant has the required form: x_1 and x_2 only occur inside an LA and with the correct coefficients in $x_1 + x_2$ while x only occurs as first parameter of an EQ term, which appears positively in the negation normal form (by the definition of LA_S and LA_W).

5.2 Pivoting of Mixed Literals

In this section we give the resolution rule for a step involving a mixed inequality $a + b \leq c$ as pivot element. In the following we denote the auxiliary variable of the negated literal $\neg(a + b \leq c)$ with x_1 and the auxiliary variable of $a + b \leq c$ with x_2. The intuition here is that x_1 and $-x_2$ correspond to the same value between a and $c - b$. The resolution rule for pivot element $a + b \leq c$ is as follows where the values for s_3, k_3 and F_3 are given later.

$$\frac{\begin{array}{c} C_1 \vee a + b \leq c : I_1[LA(c_1x_1 + s_1(\boldsymbol{x}), k_1, F_1(x_1, \boldsymbol{x}))] \\ C_2 \vee \neg(a + b \leq c) : I_2[LA(c_2x_2 + s_2(\boldsymbol{x}), k_2, F_2(x_2, \boldsymbol{x}))] \end{array}}{C_1 \vee C_2 : I_1[I_2[LA(s_3(\boldsymbol{x}), k_3, F_3)]]} \qquad \text{(rule-la)}$$

The formula $LA(s_3, k_3, F_3)$ should hold if and only if there is some $x_1 = -x_2$ such that $LA(c_1x_1 + s_1, k_1, F_1)$ and $LA(c_2x_2 + s_2, k_2, F_2)$ hold. From $c_1x_1 + s_1(\boldsymbol{x}) \leq 0$ and $c_2x_2 + s_2(\boldsymbol{x}) \leq 0$ and $x_1 = -x_2$ we get $c_2s_1(\boldsymbol{x}) + c_1s_2(\boldsymbol{x}) \leq 0$, hence we choose

$$s_3(\boldsymbol{x}) = c_2s_1(\boldsymbol{x}) + c_1s_2(\boldsymbol{x}).$$

For the inverse direction we need to guarantee the existence of $x_1 = -x_2$ between $\frac{s_2(\boldsymbol{x})}{c_2}$ and $\frac{-s_1(\boldsymbol{x})}{c_1}$ such that the following formulae hold:

$$F_1^*(x_1) :\equiv s_1(\boldsymbol{x}) + c_1x_1 \geq -k_1 \rightarrow F_1(x_1, \boldsymbol{x}),$$
$$F_2^*(x_2) :\equiv s_2(\boldsymbol{x}) + c_2x_2 \geq -k_2 \rightarrow F_2(x_2, \boldsymbol{x}).$$

In the integer case, we can guarantee this if $c_2 s_1(\boldsymbol{x}) + c_1 s_2(\boldsymbol{x}) < -c_2 k_1 - c_1 k_2 - c_1 c_2$ by choosing $x_1 = \left\lfloor \frac{-s_1(\boldsymbol{x})-k_1-1}{c_1} \right\rfloor$. Otherwise there are only finitely many candidates for $x_1 = -x_2$ between $\frac{s_2(\boldsymbol{x})}{c_2}$ and $\frac{-s_1(\boldsymbol{x})}{c_1}$. For these we can do a finite case distinction in F_3. This suggests the definitions

$$\begin{aligned} k_3 &:= c_2 k_1 + c_1 k_2 + c_1 c_2 \\ F_3(\boldsymbol{x}) &:\equiv \bigvee_{i=0}^{\left\lceil \frac{k_1+1}{c_1} \right\rceil} F_1^*\left(\left\lfloor \frac{-s_1(\boldsymbol{x})}{c_1} \right\rfloor - i\right) \wedge F_2^*\left(i - \left\lfloor \frac{-s_1(\boldsymbol{x})}{c_1} \right\rfloor\right) \end{aligned} \qquad \text{(int case)}$$

In the real case, we require that k_1 and k_2 are either $-\varepsilon$ and 0. Then, the only candidate for x_1 is $\frac{-s_1(\boldsymbol{x})}{c_1}$. We define

$$\begin{aligned} k_3 &:= \begin{cases} -\varepsilon & \text{if } k_1 = k_2 = -\varepsilon \\ 0 & \text{if } k_1 = 0 \vee k_2 = 0 \end{cases} \\ F_3(\boldsymbol{x}) &:\equiv F_1^*\left(\tfrac{-s_1(\boldsymbol{x})}{c_1}\right) \wedge F_2^*\left(-\tfrac{-s_1(\boldsymbol{x})}{c_1}\right) \end{aligned} \qquad \text{(real case)}$$

With these definition we can state the following lemma.

Lemma 2. *Let $s_1(\boldsymbol{x}), s_2(\boldsymbol{x})$ be linear terms over $\boldsymbol{x}$, $c_1, c_2 \geq 0$, $k_1, k_2 \in \mathbb{Z}$ (integer case) or $k_1, k_2 \in \{0, -\varepsilon\}$ (real case), $F_1(x_1, \boldsymbol{x}), F_2(x_2, \boldsymbol{x})$ arbitrary formulae and s_3, k_3, F_3 as defined above. Then*

$$\begin{gathered} (\exists x_1. LA_S(c_1 x_1 + s_1(\boldsymbol{x}), k_1, F_1(x_1, \boldsymbol{x})) \wedge LA_S(-c_2 x_1 + s_2(\boldsymbol{x}), k_2, F_2(-x_1, \boldsymbol{x}))) \\ \rightarrow LA_S(s_3(\boldsymbol{x}), k_3, F_3(\boldsymbol{x})) \end{gathered}$$

and

$$\begin{gathered} LA_W(s_3(\boldsymbol{x}), k_3, F_3(\boldsymbol{x})) \rightarrow \\ (\exists x_1. LA_W(c_1 x_1 + s_1(\boldsymbol{x}), k_1, F_1(x_1, \boldsymbol{x})) \wedge LA_W(-c_2 x_1 + s_2(\boldsymbol{x}), k_2, F_2(-x_1, \boldsymbol{x}))) \end{gathered}$$

This lemma can be used to show that (rule-la) is correct.

Theorem 3 (Soundness of (rule-la)). *Let $a + b \leq c$ be a mixed literal with the auxiliary variable x_2, and x_1 be the auxiliary variable of the negated literal. If $I_1[LA(c_1 x_1 + s_1, k_1, F_1)]$ yields two partial interpolants (strong and weak) of $C_1 \vee a + b \leq c$ and $I_2[LA(c_2 x_2 + s_2, k_2, F_2)]$ yields two partial interpolants of $C_1 \vee \neg(a + b \leq c)$ then $I_1[I_2[LA(s_3, k_3, F_3)]]$ yields two partial interpolants of the clause $C_1 \vee C_2$.*

To ease the presentation, we gave the rule (rule-la) with only one LA term per partial interpolant. The generalised rule requires the partial interpolants of the premises to have the shapes $I_1[LA_1^{(1)}] \dots [LA_n^{(1)}]$ and $I_2[LA_1^{(2)}] \dots [LA_m^{(2)}]$. The resulting interpolant is

$$I_1[I_2[LA_{11}^{(3)}] \dots [LA_{1m}^{(3)}]] \dots [I_2[LA_{n1}^{(3)}] \dots [LA_{nm}^{(3)}]]$$

where $LA_{ij}^{(3)}$ is computed from $LA_i^{(1)}$ and $LA_j^{(2)}$ as explained above.

6 Conclusion and Future Work

We presented a novel interpolation scheme to extract Craig interpolants from resolution proofs produced by SMT solvers without restricting the solver or reordering the proofs. The key ingredients of our method are virtual purifications of troublesome mixed literals, syntactical restrictions of partial interpolants, and specialized interpolation rules for pivoting steps on mixed literals.

In contrast to previous work, our interpolation scheme does not need specialized rules to deal with extended branches as commonly used in state-of-the-art SMT solvers to solve $\mathcal{LA}(\mathbb{Z})$-formulae. Furthermore, our scheme can deal with resolution steps where a mixed literal occurs in both antecedents, which are forbidden by other schemes [5,11].

Our scheme works for resolution based proofs in the DPLL(T) context provided there is a procedure that generates partial interpolants with our syntactic restrictions for the theory lemmas. We sketched these procedures for the theory lemmas generated by either congruence closure or linear arithmetic solvers producing Farkas proofs. In this paper, we limited the presentation to the combination of the theory of uninterpreted functions, and the theory of linear arithmetic over the integers or the reals. Nevertheless, the scheme could be extended to support other theories. This requires defining the projection functions for mixed literals in the theory, defining a pattern for weak and strong partial interpolants, and proving a corresponding resolution rule.

We plan to produce interpolants of different strengths using the technique from D'Silva et al. [8]. This is orthogonal to our interpolation scheme (particularly to the weak and strong interpolants used for mixed literals). Furthermore, we want to extend the correctness proof to show that our scheme works with inductive sequences of interpolants [15] and tree interpolants [12]. We also plan to extend this scheme to other theories including arrays and quantifiers.

References

1. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 88–102. Springer, Heidelberg (2011)
2. Bruttomesso, R., Rollini, S., Sharygina, N., Tsitovich, A.: Flexible interpolation with local proof transformations. In: ICCAD, pp. 770–777. IEEE (2010)
3. Christ, J., Hoenicke, J., Nutz, A.: Proof tree preserving interpolation. AVACS Technical Report 89, SFB/TR 14 AVACS (October 2012) ISSN: 1860-9821, `http://www.avacs.org/paper/`
4. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
5. 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)
6. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM 52(3), 365–473 (2005)

7. 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)
8. 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)
9. 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)
10. Fuchs, A., Goel, A., Grundy, J., Krstić, S., Tinelli, C.: Ground Interpolation for the Theory of Equality. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 413–427. Springer, Heidelberg (2009)
11. Goel, A., Krstić, S., Tinelli, C.: Ground Interpolation for Combined Theories. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 183–198. Springer, Heidelberg (2009)
12. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL, pp. 471–482. ACM (2010)
13. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL 2004, pp. 232–244. ACM (2004)
14. McMillan, K.L.: An Interpolating Theorem Prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (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. McMillan, K.L.: Interpolants from z3 proofs. In: FMCAD, pp. 19–27. FMCAD Inc. (2011)
17. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
18. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and Abstract DPLL Modulo Theories. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 36–50. Springer, Heidelberg (2005)
19. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. 62(3), 981–998 (1997)
20. 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)

Asynchronous Multi-core Incremental SAT Solving

Siert Wieringa and Keijo Heljanko*

Aalto University, School of Science
Department of Information and Computer Science
PO Box 15400, FI-00076 Aalto, Finland
{siert.wieringa,keijo.heljanko}@aalto.fi

Abstract. Solvers for propositional logic formulas, so called SAT solvers, are used in many practical applications. As multi-core and multi-processor hardware has become widely available, parallelizations of such solvers are actively researched. Such research typically ignores the incremental problem specification feature that modern SAT solvers possess. This feature is, however, crucial for many of the real-life applications of SAT solvers. Such applications include formal verification, equivalence checking, and typical artificial intelligence tasks such as scheduling, planning and reasoning.

We have developed a multi-core SAT solver called Tarmo, which provides an interface that is compatible with conventional incremental solvers. It enables substantial performance improvements for many applications, without requiring code modifications. We present the *asynchronous interface*, a natural extension to the conventional solver interface that allows the construction of efficient application specific parallelizations. Through the asynchronous interface multiple problems can be given to the solver simultaneously. This enables conceptually simple but efficient parallelization of the solving process. Moreover, an asynchronous solver is easier to run in parallel with other independent tasks, simplifying the construction of so called coarse grained parallelizations. We provide an extensive experimental evaluation to illustrate the performance of the proposed techniques.

1 Introduction

Propositional satisfiability (typically abbreviated SAT) is the problem of finding a satisfying truth assignment for a given propositional logic formula, or determining that no such assignment exists. This classifies the formula as respectively *satisfiable* or *unsatisfiable*. SAT is an important theoretical problem as it was the first problem ever to be proven NP-complete [9].

Despite the theoretical hardness of SAT, current state-of-the-art decision procedures for SAT, so called *SAT solvers*, have become surprisingly efficient. Subsequently these solvers have found many industrial applications. Such applications are rarely limited to solving just one decision problem. Instead, a single application will typically solve a series of related problems. Modern SAT solvers handle such problem sequences through their *incremental SAT* interface [26,11]. Using incremental SAT solvers avoids loading common subformulas over and over again. Moreover, it allows the solver to reuse

* This work was financially supported by the Academy of Finland, project 139402.

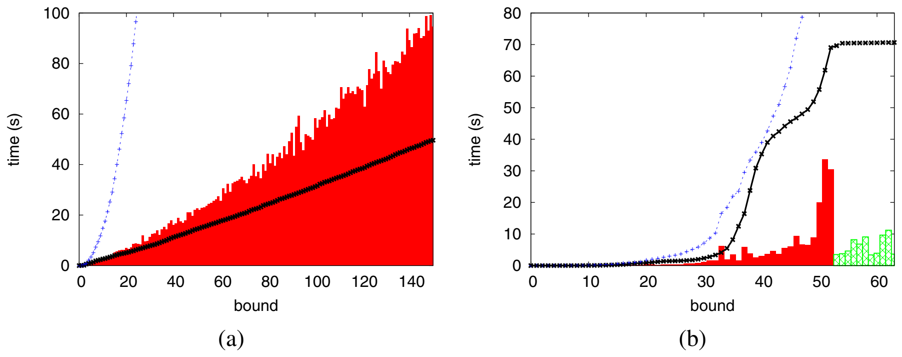


Fig. 1. Illustration of BMC run time behavior from [27]

the information it has gathered for consecutive problems. The resulting performance improvements make incremental SAT a crucial feature for modern SAT solvers.

One of the most common industrial uses of SAT solvers is in the area of formal verification. A particularly well established SAT based technique in this area is *Bounded Model Checking* (BMC) [4]. *Model checking* concerns proving temporal properties of systems, modelled e.g. as finite state machines. If a property does not hold for a system then this can be witnessed by a *counterexample*, which is a single valid execution of the system in which the property is falsified. Testing the existence of counterexamples of a bounded length can be easily done using SAT solvers. To achieve this, one defines an *unrolling function* which maps a formal system description, a temporal property, and an integer called the *bound* to a propositional logic formula. The unrolling function must encode the formula such that it is satisfiable iff a counterexample no longer than the given bound exists[1]. A typical BMC algorithm repeats this process starting from bound zero, and incrementing it by one as long as no counterexample is found.

Fig. 1 shows two illustrations of BMC run time behavior from [27], demonstrating the crucial impact of incremental SAT solving on BMC algorithm performance. The graphs illustrate solving time per bound for two different BMC benchmarks. The height of a bar in the graphs corresponds to the run time of a SAT solver on the formula for the corresponding bound without using incremental solving. The thick black curves illustrate the behavior of an incremental SAT solver that solved the formulas corresponding to all bounds sequentially, reporting its total run time each time it proceeded to the next formula in the sequence. The dotted blue curves are meant to further emphasize the poor performance of the non-incremental solver, by illustrating the cumulative run time of solving all formulas sequentially and independently.

Note that for the benchmark *eijk.S1238.S* illustrated in Fig. 1a the total run time for solving all bounds sequentially is only half that of solving the largest formula alone. Here, the gradual introduction of the problem to the solver has helped it to guide its search process, by "tuning" the solver on the smallest problems. Fig. 1b illustrates the behavior for benchmark *irst.dme6* for which the shortest counterexample is of length

[1] Another frequently used semantics is such that the formula is satisfiable iff the counterexample has a length *exactly* equal to the bound. This will be discussed in Sec. 3.4.

53. The satisfiability of the formulas for bounds larger than or equal to 53 is emphasized by the hatched bars in the figure. Although solving only one of the satisfiable formulas using a non-incremental solver would be the fastest way of establishing the existence of a counterexample there is no way of telling in advance at what bound this "easy" problem resides. Advanced heuristics [23] for such predictions will be discussed in Sec. 3. For now, observe that the incremental solver provides a robust way of finding a counterexample without previous knowledge of its length.

Despite the importance of incremental solving for practical applications SAT solvers are typically benchmarked only on single formulas, both in research publications and during SAT solver competitions[2]. The community researching a different type of constraint solvers, called SMT solvers (*Satisfiability Modulo Theories*), has acknowledged the importance of incremental solving, by introducing the *application track* to their annual competition[3]. In that track solvers are tested on incremental problems [8].

Now that multi-core and multi-processor hardware has become widely available, parallelization of SAT solvers is actively researched [5,18,31,14,15,17]. Two major approaches can be distinguished. The first is the classic divide-and-conquer approach, which aims to partition the formula to divide the total workload evenly over multiple SAT solver instances [5,24,31]. The second approach is the so called *portfolio* approach [14]. Rather than partitioning the formula, portfolio systems run multiple solvers in parallel each of which attempt to solve the same formula. The system finishes whenever the fastest solver is done. Many such portfolios consist simply of multiple instances of the same CDCL solver, as such solvers can be made to all traverse the search space in different orders by as little as using different random seeds. Portfolio solvers thus mostly exploit the run time variance of different SAT solver runs on a single formula. This approach can be surprisingly effective. Parallel SAT solvers of both types can be extended with exchange of learnt clauses between SAT solver instances, which can greatly improve the efficiency, even enabling occasional super-linear speed-ups. Both techniques are evaluated in detail in [16] and elements from both techniques are used in a recently published new technique [17,18].

To the best of our knowledge, none of the work on parallelizing SAT solvers considered maintaining the incremental features, making these parallelizations hard to apply in many practical applications. In [29] we introduced Tarmo, which at the time was only envisioned to be a special purpose parallel solver for BMC. In 2011 Tarmo competed in the Hardware Model Checking Competition (HWMCC11), where it won the new experimental multi-property and satisfiable liveness property tracks. The competing version can be seen as a parallelization of the minimalistic BMC algorithm implementation *aigbmc*[4]. The latest Tarmo version, released in October 2012, is the first version that is easy to integrate into existing applications. It can provide such applications with substantial performance improvements, without requiring them to be modified.

This work makes explicit the notion of *asynchronous* incremental SAT, a simple but crucial concept for combining incremental SAT and parallelism. It allows more efficient parallelizations of the solving process, and simplifies the construction of *multi-*

[2] `http://www.satcompetition.org`
[3] `http://smtcomp.sourceforge.net`
[4] Part of the AIGER 1.9 toolset, `http://fmv.jku.at/aiger`

engined tools. Multi-engined designs are commonly found amongst applications of SAT solvers. For example, the majority of model checkers[5] that competed at HWMCC11 fall in this category [25]. Such tools include implementations of several different algorithms (engines) over which the available computation resources are divided. Although this division can be implemented using a sequential interleaving of execution steps of the different algorithms, nowadays such tools often employ so called *coarse grained parallelization*. This means that the tools perform largely independent tasks in parallel.

A related work is *Simultaneous SAT* [19]. The interface of a simultaneous SAT solver is different from a conventional solver as for each formula in the input sequence a set of *proof objectives* can be given. This type of solver aims to prove or disprove all of these proof objectives simultaneously, i.e. in a single backtracking search. The developers of simultaneous SAT intended it to be used for BMC algorithms that check multiple safety properties per bound. Unlike our approach simultaneous SAT requires modifying the search process of the solver. Using our asynchronous interface the behavior of a simultaneous solver can be simulated, and even parallelized. A simultaneous solver with an asynchronous interface can be envisioned, but has not been investigated.

2 Incremental SAT

In order to define and discuss incremental SAT in detail this section starts with some basic definitions. A *literal* l is either a Boolean variable x or its negation $\neg x$, and double negations cancel out, hence $\neg\neg l = l$. An *assignment* is a set of literals A such that if $l \in A$ then $\neg l \notin A$. The assignment A should be interpreted such that $l \in A$ means that l is assigned the truth value **true**, and $\neg l \in A$ means that l is assigned the truth value **false**. A *clause* c is a set of literals $c = \{l_0, l_1, \cdots, l_n\}$ representing the disjunction $\bigvee c = l_0 \vee l_1 \cdots \vee l_n$. Hence, clause c is *satisfied* by assignment A iff $l \in A$ for some $l \in c$. Moreover, a clause consisting of exactly one literal is called a *unit clause*. A *cube* d is a set of literals $d = \{l_0, l_1, \cdots, l_n\}$ representing the conjunction $\bigwedge d = l_0 \wedge l_1 \cdots \wedge l_n$. Hence, cube d is *satisfied* by assignment A iff $d \subseteq A$.

A formula is in *Conjunctive Normal Form (CNF)* if it is a conjunction of disjunctions, i.e. a set of clauses. A CNF formula is satisfied by an assignment that satisfies all of its clauses. A formula for which such a *satisfying assignment* exists is *satisfiable*, other formulas are *unsatisfiable*. Conventional SAT solvers handle only CNF formulas.

The most commonly used SAT solvers are of the *Conflict Driven Clause Learning* (CDCL) type [21]. Such solvers derive new clauses, called *learnt clauses*, during their solving process. These learnt clauses are logical consequences of the clauses in the input formula, and their derivation is intended to help the solver avoid parts of the search space that are without satisfying assignments. In this work the term solver always refers to a CDCL SAT solver for CNF formulas.

A general definition for the incremental satisfiability problem is given in [26], where it is defined as solving each formula in a finite sequence of formulas. The transformation from a formula to its successor in the sequence is defined by two sets, a set of clauses to be added and a set of clauses to be removed. Although it is possible to implement

[5] e.g. ABC [7] `http://www.eecs.berkeley.edu/~alanmi/abc`
and PdTRAV `http://fmgroup.polito.it/quer/research/tool/tool.htm`

a SAT solver that allows arbitrary removal of clauses between consecutive formulas, there is a complication in that when a clause is removed also all learnt clauses whose derivation depends on that clause must be removed. Maintaining sufficient information in the solver to achieve this has significant drawbacks on its performance and thus arbitrary clause removal is not implemented in any state-of-the-art solver.

Multiple solutions exists. For example, in the interface of the SAT solver *zChaff*[6], an implementation of the *chaff* algorithm [22], it is possible to assign clauses to groups, and those groups can be removed as a whole. The SMT-LIB standard [3] for SMT solver input defines the so called push- and pop-interface. In this approach the subproblems are maintained on a stack and the solver aims to solve the union of the problems on that stack. The simplest and most commonly used interface for incremental SAT solvers however is the one defined in [11] and first used in the solver *MiniSAT* [10]. This solver interface does not contain a function for removing clauses. Instead, a solver with this interface can determine the existence of satisfying assignments that include a specified set of *assumptions*. The interface is defined by two functions:

- `addClause(Clause clause)`
- `solve(Cube assumptions)`

Using this interface clause removal can be simulated as follows: Instead of adding clause c to the solver the clause $c \cup \{x\}$ where x is a free variable is added. As long as the solver is asked to perform its solving task under a set of assumptions that includes literal $\neg x$ it will only consider assignments A such that $\neg x \in A$, hence it must satisfy c in order to satisfy clause $c \cup \{x\}$. However, without the assumption $\neg x$ the solver can assign x to **true** and ignore c.

Note that the `addClause` and `solve` function define part of the interface of a SAT solver, hence they control the execution of this particular computer program. The `solve` function is *blocking*, in the sense that the call to this function will not return to the calling application until the SAT solver determines the satisfiability of the loaded problem. In this work the input for an incremental SAT solver is defined separately from the execution of such a solver. Here, an instance of the incremental SAT problem is defined as a sequence of *jobs* $\langle \phi_0, \phi_1, \cdots \rangle$. A job ϕ_i is characterized by a set of clauses $\texttt{CLS}(\phi_i)$ and a single cube $\texttt{assumps}(\phi_i)$. Each job ϕ_i induces a CNF formula $\mathcal{F}(\phi_i)$ consisting of all its clauses and all clauses in previous jobs, and one unit clause for each literal in its cube of assumptions.

$$\mathcal{F}(\phi_i) = \underbrace{\left(\bigcup_{0 \le j \le i} \texttt{CLS}(\phi_j) \right)}_{\texttt{CLAUSES}(\phi_i)} \cup \left(\bigcup_{l \in \texttt{assumps}(\phi_i)} \{l\} \right)$$

In the rest of this work "solving a job" refers to the process of determining the satisfiability of the CNF formula induced by that job. Note that these definitions have been chosen to match solvers using the interface of [11]. Calling `addClause(`c`)` for all

[6] `http://www.princeton.edu/~chaff/zchaff.html`

$c \in \texttt{CLAUSES}(\phi_i)$ followed by a call to `solve(assumps(`ϕ_i`))` will make such solver solve $\mathcal{F}(\phi_i)$ (assumptions are handled as truth assignments in the solver).

Without enforcing the blocking semantics of the `solve` function it is possible to think of the solver as a reactive system. The system is given jobs as input and as output it reports the result of solving those jobs. The communication between the application and the solver is *asynchronous*: The application may proceed to submit more jobs while the solver has not yet reported the result for a previously submitted job. Moreover, the results may be reported by the solver out-of-order with respect to the order of the jobs in the input sequence.

3 Employing Asynchronicity and Parallelism

To motivate the *asynchronous* communication between application and solver proposed in the previous section let us take another look at Fig. 1b. Note that the largest unsatisfiable formulas, those for bounds just below 53, are much harder to solver than the smallest satisfiable ones. It was observed in [27] that this type of run time profile is typical for formula sequences from BMC that contain satisfiable formulas. This matched earlier observations [23] for a different application of SAT solvers called *automated planning*. In automated planning the satisfiability of a formula in the sequence corresponds to the existence of a *plan* of a certain length. The two applications are similar in nature: Either all formulas in the sequence are unsatisfiable, or the sequence has a finite prefix of formulas that are unsatisfiable, followed by only satisfiable formulas.

The authors of [23] did not consider incremental solving, but rather aimed to improve the speed at which the existence of a satisfiable formula in the sequence can be established using a non-incremental solver. They suggested that instead of always aiming to solve the first unsolved formula in the sequence, the total solving effort can be divided over a prefix of the unsolved formulas in the sequence. Under the observed typical run time profile this would then allow solving a satisfiable formula before the solving of the hardest unsatisfiable formulas has been completed. This is an interesting idea, but without the use of an incremental solver it is handicapped especially on long subsequences of unsatisfiable formulas. Although dividing the effort over multiple formulas can be beneficial, it is not useful if the extra performance provided by the incremental solver is lost. Asynchronicity provides a way to give an incremental solver any prefix of the formula sequence rather than just one formula at the time.

3.1 Parallelizing Incremental SAT

The algorithms used in parallel SAT solvers for doing the actual solving are often identical to those used in sequential solvers. A typical parallel SAT solver's architecture uses multiple conventional sequential solvers in parallel. In portfolio solvers these parallel operating solvers are all given the same input, whereas in other approaches each solver instance is restricted to a portion of the search space. The basic building block in our parallel incremental SAT solver called Tarmo is a conventional incremental SAT solver using the assumptions interface, currently MiniSAT 2.2[7]. During its execution Tarmo

[7] http://www.minisat.se

spawns multiple *solver threads*, and each of these threads has access to its own instance of the conventional solver. Tarmo's interface is similar to that of any other SAT solver, except that it provides two extra non-blocking functions called `addCube` and `cancel`. The `addCube` function enters an assumptions cube, and thereby induces a new job in the sequence of jobs stored inside Tarmo. Each of its solver threads repeatedly reads a job from the sequence and solves it. The `cancel` function can be used to cancel the solving of a specific job.

If all of the solver threads always read the first unsolved job from the sequence then Tarmo becomes a portfolio of incremental solvers, e.g. each solver thread tries to solve all of the jobs in the input sequence. We refer to this strategy as *distribution mode* `multiconv` (multiple conventional). In a different distribution mode of Tarmo, called `multijob`, each of the solver threads always proceeds to solve the first unsolved job from the sequence that has not yet been assigned to another solver thread. This matches the natural idea that for an efficient parallelization the work performed by the separate threads should be different. This strategy was also used by a parallel solver specifically designed around one BMC unrolling function [1]. The `multijob` strategy does have a downside: Each solver thread individually no longer solves all of the jobs, hence the individual benefit of incremental solving is reduced.

As the solver threads use conventional incremental solvers no clauses can be removed by the solver threads. As a consequence, Tarmo can only use distribution modes which are defined such that a thread which just solved ϕ_i can only proceed to solve ϕ_j if $\texttt{CLAUSES}(\phi_i) \subseteq \texttt{CLAUSES}(\phi_j)$. Note that it is possible that $\texttt{CLAUSES}(\phi_i) = \texttt{CLAUSES}(\phi_j)$ for $i \neq j$ because applications may test the same set of clauses under different sets of assumptions. In such cases there are jobs ϕ_j such that $\texttt{CLS}(\phi_j) = \emptyset$. For example, in Cube-And-Conquer [15], one set of clauses is tested under many thousands of different sets of assumptions.

3.2 Clause Sharing

Sharing of learnt clauses is an important building block in any parallel SAT solver. Although sharing learnt clauses between different solver threads can allow those threads to help each other, sharing too many clauses harms performance. Even conventional sequential solvers do not store all the learnt clauses they derive forever, but rather they clean up their learnt clause database regularly during the solving process. Restricting the number of learnt clauses shared between solving threads is therefore an important aspect of parallel SAT solving (see, e.g. [13]). It was stated in the introduction that incremental SAT solving "allows the solver to reuse the information it has gathered for consecutive problems". The learnt clauses are an important part of this information, although some heuristics measures kept in the solver are also important [27].

The asynchronous interface allows solving multiple jobs in any order. In particular, in Tarmo, multiple solver threads may not be solving the same job at the same time. Hence, care must be taken when employing sharing of learnt clauses between those solver threads. Note that in general a clause c derived while solving a job ϕ_i can be used in the solving process of any job ϕ_j such that $\texttt{CLAUSES}(\phi_i) \subseteq \texttt{CLAUSES}(\phi_j)$.

To achieve correct clause sharing with low overhead the database in Tarmo is organized as a set of queues. There is one queue for each unique clause set, i.e. one queue

$q(\phi_i)$ for each job ϕ_i such that $\texttt{CLS}(\phi_i) \neq \emptyset$. For jobs ϕ_j such that $\texttt{CLS}(\phi_j) = \emptyset$ we have $q(\phi_j) = q(\phi_i)$ for the largest i such that $i < j$ and $\texttt{CLS}(\phi_i) \neq \emptyset$. If a solver thread wants to share a learnt clause it derived while working at job ϕ_i it pushes it in the corresponding queue $q(\phi_i)$. A solver thread that is solving ϕ_j can now safely read and enter any foreign learnt clause that it can find in the queues $q(\phi_i)$ for all $i \leq j$.

The number of learnt clauses stored in each of the solver threads, and thus nominated for sharing with others, is not as massive in Tarmo as in conventional parallel SAT solvers for three different reasons. In Tarmo the solver threads only read and write to the queues in the shared clause database at the start and end of a job, and during *restarts* [12]. Some conventional solvers use a much more eager strategy. Sharing only at restarts however has the nice property that the introduction of new learnt clauses does not interfere with active search processes. The second reason is that the formulas used to test conventional parallelizations of SAT solvers are usually amongst the hardest its developer can find. Tarmo instead deals with sequences of problems for which the difficulty is typically more in the length of the sequence than in the hardness of individual formulas. The third reason is more implementation specific, but related to the second one. SAT solvers use a limit on the number of learnt clauses they store in their databases, and as the search continuous they increase this limit. A specific feature of MiniSAT, and thus also of the solving threads in Tarmo, is that when incremental solving is used this limit is reset for every consecutive call to `solve`. Hence, compared to solving a single hard instance for the same amount of time the clause database grows less large on an incremental problem sequence. During experiments for [15] this was found to be a crucial element in MiniSAT's incremental solving performance.

Unlike the common wisdom regarding conventional parallel SAT solvers, a version of Tarmo that shares *all* learnt clauses performs substantially better than the version that shares no clauses at all. Limiting the throughput of learnt clauses does improve its performance further, especially for harder problems. Tarmo limits the sharing of learnt clauses on the sending side only, i.e. clauses that are not considered of sufficient "quality" are not placed into the queues of the shared clause database. Two measures of clause quality that can be determined quickly are their length, and their Literals Blocks Distance (LBD) [2]. Because shorter clauses represent stronger constraints limiting the length of shared clauses by a constant (8 in [14]) would be a reasonable and very simple heuristic. The problem is that as the search continues the length of the clauses tends to increase, reducing the throughput of shared clauses [13]. Tarmo therefore by default shares all clauses whose length is below the running average, and this default is used in all results presented in this work. It is possible to configure Tarmo to share clauses below the average (or a constant) LBD, but this does not improve the average performance for the experiments presented here. The result of the experiments for different clause sharing heuristics can be found from the authors' webpage[8].

3.3 The Synchronous Interface: A Drop-in Replacement for MiniSAT

The aim of our work is to provide performance improvements for applications of incremental SAT solvers, without requiring extensive rewriting of those applications.

[8] `http://users.ics.aalto.fi/swiering/tacas13`

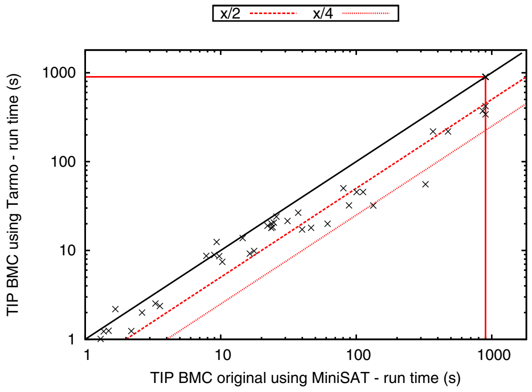


Fig. 2. Replacing MiniSAT by Tarmo without further modifications

To illustrate that this can be achieved we took the latest version of the model checker TIP[9] and replaced the MiniSAT solver with Tarmo. Tarmo's interface provides a blocking `solve` call for full source-code compatibility with MiniSAT. Because of this compatible interface the modification of the source code of TIP was limited to just changing the name of the type of the solver. Although an application that uses Tarmo as a drop-in replacement for MiniSAT does not benefit from asynchronicity directly, it can still benefit from parallelism. Through Tarmo, and the `multiconv` distribution mode it provides, the application now has access to a portfolio of incremental solvers that are performing learnt clause sharing. Because most popular SAT solvers other than MiniSAT also use MiniSAT-like interfaces, replacing such solvers by Tarmo in existing applications should not be much harder.

All experiments in this work were performed in a computing cluster in which each node has two six core Intel Xeon X5650 processors. A memory limit of 3500MB per solver thread was employed. Fig. 2 is a logarithmic-scale scatterplot that shows the performance of the proposed straightforward use of Tarmo for the BMC algorithm inside TIP. This experiment was performed using the 95 benchmarks from the single safety property track of HWMCC11 for which during the competition at least one model checker found a counterexample. The version of TIP using the original MiniSAT solver solved 84 of those benchmarks within 900 seconds. By using Tarmo with 4 solver threads instead the performance of TIP is improved enough to make it solve 86 benchmarks. For the 24 benchmarks that were solved by the unmodified version of TIP in more than 10 seconds, an average speed-up of 2.1 is obtained by using Tarmo. A two time speed-up using four times the number of solver threads is not bad, considering that each of the solver threads are solving the exact same sequence of problems. During this experiment each of the solver threads used the exact same settings, except for the random seed. It should be possible to further increase the performance by using a variety of different settings for each solver thread, but this would require an extensive empirical evaluation that is outside the scope of this paper. The surprising strength of this approach matches observations for conventional parallel SAT solvers [14,18].

9 `http://github.com/niklasso`

3.4 The Asynchronous Interface: Exploiting Application Specific Knowledge

The asynchronous incremental solver interface is a natural extension to a basic incremental solver and can prove useful for many applications. Exploiting it effectively does however require some knowledge of the application.

The sequence of formulas generated from applications like BMC or automated planning can be generated up to any arbitrary length in advance. This does not hold for many other applications of incremental solvers in which the encoding of formulas depends on the results of solving previous formulas.

The main loop of a conventional BMC algorithm, as found in TIP, is given in Fig. 3a. The BMC unrolling function, providing the transition relation of a system for a bounded number of steps in propositional logic, is named `unroll` in the pseudocode. For this work it suffices to understand `unroll` as a function that makes repeated calls to the solver's `addClause` function and then returns a set of assumption literals. Once this has been done the `solve` function is called to establish the satisfiability of all clauses under the set of assumptions. If the solver finds this satisfiable then a counterexample of length k has been found, otherwise the value of k is incremented and the next iteration of the loop starts.

Fig. 3b illustrates a BMC loop exploiting the asynchronous solver interface. The non-blocking function `addCube` is called after `unroll`, inducing job ϕ_k for the solver. Note that $\mathcal{F}(\phi_k)$ is exactly the same formula that would have been solved in iteration k of the conventional algorithm. On the Lines I-III the actions that must be executed when a result is received from the solver are stated. This result handling code can be executed in a thread concurrent to the thread executing the main loop, or alternatively it can be handled by the same thread if a poll to the solver for new results is included in the loop. In either case, Tarmo reports a result for each job ϕ_i at most once. For all but the most trivial benchmarks the encoding of a formula using the `unroll` function can be performed much faster than solving that formula. Hence, to avoid wasting large amounts of memory, in practice it is necessary to limit the number of unsolved jobs in the solver to a small constant. To illustrate this in Fig. 3b on Line 7 the job generation is paused until the value of shared variable p falls below constant value $max\_pending$. Alternatively, such limits can be implemented using functions provided by the interface of Tarmo, avoiding the need to handle potential concurrency issues in the application.

We modified TIP to use asynchronous BMC. TIP is a complex piece of software, which provides several different verification algorithms and performs non-trivial reductions on its input models. The modifications to the existing code of TIP made to introduce asynchronous BMC were, however, not more complicated than those given in Fig. 3. The performance is illustrated using a cactus plot in Fig. 4. The benchmarks used for the illustrated experiment are the same as discussed in Sec. 3.2. The two synchronous versions 'Sync. 1' and 'Sync. 4' correspond to the two algorithm versions compared in Fig. 2. Observe that using 4 solver threads and Tarmo's `multijob` distribution mode, asynchronous BMC is able to solve 88 of the benchmarks. Using 6 threads this further increases to 89, but it then goes back to 88 for the version that uses 8 threads.

Earlier in this work, and in the related work on automated planning [23], only sequences were considered that either consist only of unsatisfiable formulas, or of a finite

Conventional BMC

```
1. $k = 0$
2. forever do
3.     $A$ = unroll(k)
4.     $r$ = solve(A)
5.     if $r$ = **unsatisfiable** then
6.         $k++$
7.     else
8.         return cex of length $k$
```

(a)

Asynchronous BMC

```
1. $k = 0$; $p = 0$
2. while cex not found
3.     $p++$
4.     $A$ = unroll(k)
5.     addCube(A)
6.     $k++$
7.     wait until $p < max\_pending$
On result for job $\phi_i$:
I.   $p--$
II.  if result for $\phi_i$ is **satisfiable** then
III.     return cex of length $i$
```

(b)

Fig. 3. Pseudocode for usage of incremental SAT in BMC

prefix of unsatisfiable formulas followed by only satisfiable formulas. This means that the result handling functions of Fig. 3b can be extended with an extra application specific improvement: If the result unsatisfiable is reported then the solver may be asked to abort solving all unsolved older jobs, as these are now known to be unsatisfiable. The `cancel` function in Tarmo's interface is provided for this purpose. Unfortunately there is a problem when applying this idea in TIP, which is that for safety properties it encodes the k-th formula with the semantics that it is satisfiable iff a counterexample of *exactly* length k exists. Hence, in TIP, the unsatisfiability of a job does not necessarily imply that all older jobs are also unsatisfiable.

This problem was resolved by making a small modification to each of the benchmarks before giving them as input to our asynchronous BMC version of TIP. The benchmarks are encoded in the AIGER-format[10], a representation of Boolean circuits using and-gates, inverters and latches. Here, a counterexample is a sequence of truth assignments to the inputs of the circuit that makes the output attain the value **true**. For each benchmark a new circuit was created by extending the original circuit with a small amount of extra logic, including one latch. The added logic makes sure that, iff the output of the original circuit attains the value **true**, then the output of the new circuit attains the value **true** and remains in this state regardless of changes to the input signals. By using these modified circuits, instead of the original models, older jobs can now be safely cancelled by the asynchronous BMC result handling function. The resulting performance is shown in Fig. 5. Clearly, cancelling of older unsatisfiable jobs improves the performance and especially the scaling of the parallelization.

3.5 Coarse Grained Parallelization

For computationally hard problems, such as SAT solving or model checking, there are no "one size fits all" solutions. Because different algorithms work well for different

[10] `http://fmv.jku.at/aiger`

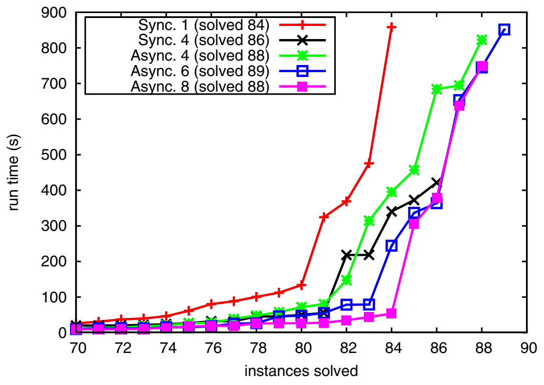


Fig. 4. TIP BMC using asynchronous solving

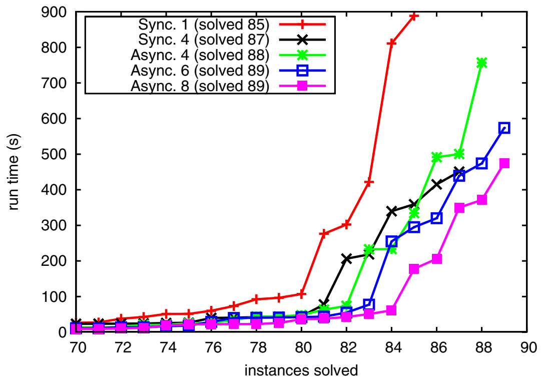
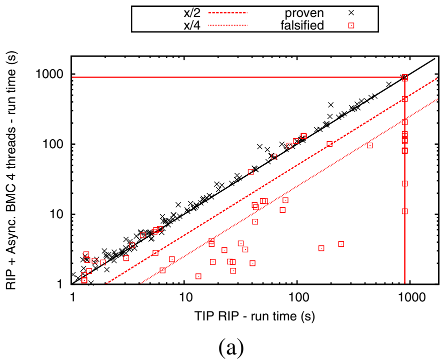


Fig. 5. TIP BMC using asynchronous solving on modified circuits

problems, tools implementing more than one algorithm, so called *algorithm portfolios*, or *multi-engined tools*, are common practice (e.g. [30,25]). Although the asynchronous interface was developed to allow parallelization of incremental SAT solving, it can also aid the development of multi-engine tools. Once again, we use TIP to illustrate our point. TIP includes an implementation of the IC3 algorithm [6] which is called the *Recursive Induction Prover (RIP)*. In contrary to the basic BMC implementation this algorithm can prove that a property holds. Although IC3/RIP can also find counterexamples it can typically not match the performance of BMC at this task, thus executing both algorithms in a portfolio should provide better average performance.

Creating such a portfolio inside TIP was easy, as we had asynchronous BMC already in place. We simply added calls to the BMC algorithm functions `unroll` and `addCube` (recall Fig. 3b) inside the main loop of the RIP algorithm. As a result, the RIP algorithm ensures the concurrent execution of the completely independent asynchronous BMC algorithm. In this set-up Tarmo is only used for BMC. Using the RIP algorithm 346 out of the 465 single safety property benchmarks from HWMCC11 can be solved within 900 seconds. By executing BMC concurrently with RIP this increases



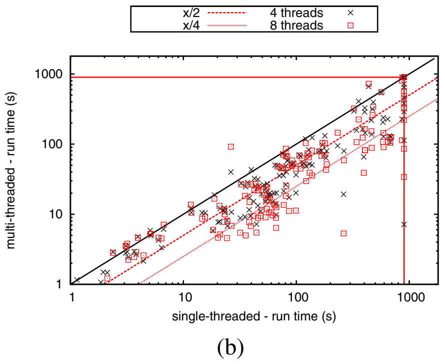


Fig. 6. Tarmo for concurrent BMC and RIP, and Tarmo for MUS Finding

to 352, with Tarmo configured to use one solving thread for BMC. Using 4 solving threads 357 benchmarks are solved. The impressive and consistent speed-up for counterexample finding is illustrated in Fig. 6a for the version using 4 solving threads.

It must be noted that simultaneous execution of two completely separated implementations of BMC and RIP as two different processes will give roughly the same performance. This experiment is only meant to illustrate that an asynchronous solver is easy to run "on the side". This clearly can have advantages over execution in separate processes. For example, one could implement a tool in which the BMC and RIP algorithms share derived system invariants, or lower-bounds on counterexample length.

3.6 Asynchronous Solving Outside BMC

Some applications of incremental solvers, such as Cube-And-Conquer [15] parallelize naturally, whereas others are very challenging. Dependencies between the generation of jobs and the result of previous jobs can make running multiple jobs concurrently harder. In this section we discuss a particularly challenging application.

An unsatisfiable CNF formula is *minimal unsatisfiable* if removing any of its clauses makes it satisfiable. Algorithms that find Minimal Unsatisfiable Subsets (MUSes) of unsatisfiable formulas have received a lot of research interest in recent years. An important recent contribution is *model rotation* [20]. The performance of that algorithm was studied in [28], which also proposed parallelization using Tarmo. This a challenging application because the concurrently executed jobs are not independent. In this parallelization the result of a job can imply that the result of concurrently solved jobs is no longer interesting. Fig. 6b shows results for a new implementation of the existing parallelization from [28]. The new implementation is based on the same ideas but benefits from Tarmo's recent interface improvements, as well as from better MUS finding heuristics. The set of benchmarks used were the 178 benchmarks also used in [28] and 34 from [4]. The single threaded version solved in total 168 benchmarks, requiring on average 2468 jobs per benchmark. The versions using 4 and 8 threads both solve 174 benchmarks. However, the 4 threaded version opportunistically generates an average of 3610 jobs per benchmark out of which only 2499 (69%) have a result that progresses the MUS finding. For the 8 threaded version only 2535 (52%) out of 4842 jobs per benchmark are effective. Despite the large amount of unnecessary work performed, this parallelization improves the performance of a state-of-the-art MUS finding algorithm.

4 Conclusions

In this paper we discussed the *asynchronous* interface for incremental SAT solvers. The incremental feature of modern SAT solvers is crucial for their performance in practical applications. Nevertheless, it is often overlooked in research aiming at improving or parallelizing such solvers. By extending the most commonly used incremental solver interface our parallelizations are directly applicable in many different contexts. As a result, substantial performance gains can be obtained by simply replacing a sequential incremental solver by our source-code compatible multi-core solver. In many cases further improvements are possible by using the asynchronous interface to create an application specific parallelization. The minimal nature of the proposed extension to the standard interface means that asynchronicity does not have to be limited to our Tarmo solver. Instead, it can prove useful to any solver developer interested in combining incremental SAT solving and parallelism.

Acknowledgements. The authors would like to thank Niklas Sörensson for providing TIP, Niklas Eén and Armin Biere for being sources of motivation and inspiration, and Matti Niemenmaa for his contributions to early versions of Tarmo.

The Tarmo solver, the modified versions of TIP, and more experimental data can be found from http://users.ics.aalto.fi/swiering/tacas13.

References

1. Ábrahám, E., Schubert, T., Becker, B., Fränzle, M., Herde, C.: Parallel SAT Solving in Bounded Model Checking. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS and PDMC 2006. LNCS, vol. 4346, pp. 301–315. Springer, Heidelberg (2007)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI, pp. 399–404 (2009)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard version 2.0 (2010), http://www.smtlib.org
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS/ETAPS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
5. Böhm, M., Speckenmeyer, E.: A fast parallel SAT-solver - efficient workload balancing. Ann. Math. Artif. Intell. 17(3-4), 381–400 (1996)
6. Bradley, A.R.: k-step relative inductive generalization. CoRR abs/1003.3649 (2010)
7. Brayton, R., Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010)
8. Bruttomesso, R., Griggio, A.: Broadening the scope of SMT-COMP: the application track. In: COMPARE, pp. 18–27 (2012)
9. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC, pp. 151–158. ACM (1971)

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. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4), 543–560 (2003)
12. Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reasoning 24(1/2), 67–100 (2000)
13. Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel SAT solving. In: Boutilier, C. (ed.) IJCAI, pp. 499–504 (2009)
14. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: A parallel SAT solver. JSAT 6(4), 245–262 (2009)
15. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 50–65. Springer, Heidelberg (2012)
16. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning Search Spaces of a Randomized Search. In: Serra, R., Cucchiara, R. (eds.) AI*IA 2009. LNCS, vol. 5883, pp. 243–252. Springer, Heidelberg (2009)
17. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Grid-Based SAT Solving with Iterative Partitioning and Clause Learning. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 385–399. Springer, Heidelberg (2011)
18. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search. Fundam. Inform. 107(2-3), 289–311 (2011)
19. Khasidashvili, Z., Nadel, A., Palti, A., Hanna, Z.: Simultaneous SAT-Based Model Checking of Safety Properties. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 56–75. Springer, Heidelberg (2006)
20. Marques-Silva, J., Lynce, I.: On Improving MUS Extraction Algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 159–173. Springer, Heidelberg (2011)
21. Marques-Silva, J.P., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD, pp. 220–227 (1996)
22. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC, pp. 530–535 (2001)
23. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artif. Intell. 170(12-13), 1031–1080 (2006)
24. Schubert, T., Lewis, M.D.T., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. JSAT 6(4), 203–222 (2009)
25. Sterin, B., Een, N., Mishchenko, A., Brayton, R.: The benefit of concurrency in model checking. In: IWLS, pp. 176–182 (2011)
26. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE: A new incremental satisfiability engine. In: DAC, pp. 542–545 (2001)
27. Wieringa, S.: On incremental satisfiability and bounded model checking. In: Ganai, M.K., Biere, A. (eds.) DIFTS, pp. 46–54 (2011)
28. Wieringa, S.: Understanding, Improving and Parallelizing MUS Finding Using Model Rotation. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 672–687. Springer, Heidelberg (2012)
29. Wieringa, S., Niemenmaa, M., Heljanko, K.: Tarmo: A framework for parallelized bounded model checking. In: Brim, L., van de Pol, J. (eds.) PDMC. EPTCS, vol. 14, pp. 62–76 (2009)
30. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. CoRR abs/1111.2249 (2011)
31. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. 21(4), 543–560 (1996)

Model-Checking Iterated Games*

Chung-Hao Huang[1], Sven Schewe[2], and Farn Wang[1,3]

[1] Graduate Institute of Electronic Engineering, National Taiwan University
[2] Department of Computer Sciences, University of Liverpool
[3] Department of Electrical Engineering, National Taiwan University

Abstract. We propose a logic for the definition of the collaborative power of groups of agents to enforce different temporal objectives. The resulting *temporal cooperation logic* (*TCL*) extends ATL by allowing for successive definition of strategies for agents and agencies. Different to previous logics with similar aims, our extension cuts a fine line between extending the power and maintaining a low complexity: model-checking TCL sentences is EXPTIME complete in the logic, and fixed parameter tractable for specifications of bounded size. This advancement over non-elementary logics is bought by disallowing a too close entanglement between cooperation and competition. We show how allowing such an entanglement immediately leads to a non-elementary complexity. We have implemented a model-checker for the logic and shown the feasibility of model-checking on a few benchmarks.

1 Introduction

While the verification of traditional linear and branching time logics like LTL, CTL, and CTL* [17,8] has been reduced to (repeated) reachability [11,13], the satisfiability checking and synthesis problem has been tightly linked with game theory ever since the seminal works of Büchi and Landweber [5,4]. With the introduction of *alternating time logic* (*ATL*) by Alur, Henzinger, and Kupferman [1] and in automata based μ-calculus model-checking (e.g., [22]), games have entered into the verification of the correctness of reactive systems. With game theoretic challenges moving into the focus of researchers who study the specification and design of reactive systems, traditional problems of multi-player games are replacing the former distinction between an adversarial environment and a supportive system. Instead, we have groups of players that cooperate on some objectives while competing on others.

For particular properties, the intuition that some players represent the system while other players represent the environment is, however, still useful. Following this intuition, the system wins the game in an execution (or a *play* in the jargon of game theory) if the system specification is fulfilled along it, and it wins the game if it can force a winning play. System design as a whole for specifications in game logics can rather be compared to designing a game board and to show that the respective group of players (or: agency) has the coalition power required by the system specification.

* The research was supported by the National Science Council grant 97-2221-E-002-129-MY3 and by the Engineering and Physical Sciences Research Council grant EP/H046623/1.

There are various established game-based specification languages, including *ATL*, *ATL**, the *alternating* μ*-calculus* (*AMC*), and *game logic* (*GL*) [1], *strategy logics* [7,9,15,14], *coordination logic* [10], *stochastic game logic* [3], and *basic strategy interaction logic* (*BSIL*) [21] for the specification of the interplay in open systems. Each language also comes with a verification algorithm that determines whether a winning strategy for the system exists. However, there is a gap between the available techniques and the scalability required for industrial applications. Frankly speaking, none of the languages above represents, in our view, a proper combination of expressiveness for close interaction among agent strategies and efficiency for the verification or refutation of compliance with a specification. On one hand, logics like ATL, ATL*, AMC, and GL [1] allow us to specify the collaborative power of groups of players to enforce a common objective. This falls short from specifying even the simple properties in a typical game. For example, it was shown in [21] that ATL, ATL*, AMC, and GL [1] cannot express that the same strategy of a banking system must allow the clients both, to withdraw and to deposit money: a strategy quantifier in these logics always refers to the strategies of all agents, whereas this property requires to bind first the strategy of the bank, and then refer to different strategies of the clients. This is arguably a severe restriction when reasoning about real-world problems.

To solve the expressiveness problem in the above example, *strategy logics* (*SL*) were proposed in [3,7,15,14]. They allow for the flexible quantification over strategies in logic formulas. However, their verification complexity is prohibitively high and has inhibited practical application.

A previous attempt to tame the complexity of strategy interaction [21], on the other hand, results in a full temporalisation. This leads to severe restrictions in the entanglement between temporal operators and strategy binding and thus prevents, for example, reasoning about Nash equilibria.

We thus propose to adapt the logic introduced in [21] to a new temporal logic called *temporal cooperation logic* (*TCL*) for this purpose. Let us introduce TCL informally on a game among three prisoners.

Example: Iterated Prisoners' Dilemma. Inspired by the famous prisoners' dilemma, we consider a model where three suspects, who are initially in custody, are interrogated. In our simplified version, they play in turns (rather than concurrently), and have the choices to either admit or deny the charges made against them. If all deny, they will be released based on lack of evidence.

However, a suspect may decide to collaborate with the police and betray her peers. A sole collaborator will be acquitted as a crown witness, while her peers will be sentenced. But if two or more suspects collaborate with the police, all will be sentenced.

In an iterated prisoners' dilemma, the interplay can continue up to an unbounded number of times. Such a game is very useful in modelling collaboration and competition in networks. For example, a strategy in prisoners' dilemma is *nice* if it does not suggest betrayal initially and only suggests betrayal if, in the previous round, another prisoner betrayed [2]. The following TCL sentence states that Prisoner 1 has a nice strategy.

$$\langle 1\rangle\Box((\langle +\rangle \bigcirc \neg\mathtt{betray}_1) \vee \textstyle\bigvee_{a\neq 1} \mathtt{betray}_a) \tag{A}$$

$\langle 1 \rangle$ is a *strategy quantifier* (SQ), which states that there exists a strategy of Prisoner 1 to achieve her temporal goal. $\langle + \rangle$ is a *strategy interaction quantifier* (*SIQ*) that inherits the strategy from its parent formula. Proposition $\mathtt{betray}_i$ is an atomic proposition for the betrayal of prisoner i at the present state. Similarly, we can reflect more involved strategies, such as 'Prisoner 2 will always betray when she does not have the power to force Player 1 to always play nice.'

$$\langle 2 \rangle((\langle + \rangle \Box \mathtt{betray}_2) \vee \langle + \rangle \Box((\langle + \rangle \bigcirc \neg \mathtt{betray}_1) \vee \textstyle\bigvee_{a \neq 1} \mathtt{betray}_a)) \tag{B}$$

Similar properties can be used to specify *forgiving*[1] or other related strategies [2]. A forgiving strategy of Prisoner 1 is reflected by the following TCL property.

$$\langle 1 \rangle \Diamond((\langle + \rangle \bigcirc \neg \mathtt{betray}_1) \wedge \textstyle\bigvee_{a \neq 1} \mathtt{betray}_a) \tag{C}$$

We can also reason about the existence of Prisoner 2's strategy that avoid betrayal if Prisoner 1 can be unforgiving under this strategy.

$$\langle 2 \rangle((\langle + \rangle \Box \neg \mathtt{betray}_2) \vee \langle +1 \rangle \Diamond((\langle + \rangle \bigcirc \neg \mathtt{betray}_1) \wedge \textstyle\bigvee_{a \neq 1} \mathtt{betray}_a)) \tag{D}$$

As can be seen, properties like (B) and (D) are relevant in network environments where plays can be extended round by round without termination. Every agent may track each others' records to decide whether or not to cooperate. Such a property cannot be expressed in ATL*, GL, AMC, or BSIL. While it can be expressed with SL, the verification complexity of SL is prohibitive.

In [21], SIQs can neither override nor revoke strategies assigned by the SQ or SIQs in whose scope they are. Consequently, BSIL cannot express deterministic Nash equilibria. To overcome this restriction, we introduce a strategy reset operator that revokes previous strategy assignments.

Let $\mathtt{jail}_a$ be a proposition, which states that "Prisoner a is in jail". In TCL,

$$\langle 1, 2, 3 \rangle \textstyle\bigwedge_{a \in [1,3]} \left((\langle + \emptyset \rangle \Diamond \neg \mathtt{jail}_a) \vee \langle -a \rangle \Box \mathtt{jail}_a\right) \tag{E}$$

requires that the tree agents can cooperate such that every agent either eventually leaves prism, or stays for ever in prism regardless of her own strategy under the current strategies of the remaining prisoners. The SIQ $\langle -a \rangle \psi$ revokes the binding of agent a to her strategy.

In this work, we establish that TCL is incomparable with ATL*, GL, and AMC in expressiveness. Although the strategy logics proposed in [3,7,9,15] subsume TCL with their flexible quantification of strategies and binding to strategy variables, their model-checking complexities are all doubly exponential time hard. In contrast, TCL enjoys an EXPTIME-complete model-checking complexity and fixed parameter tractability when using the length of the formula as parameter, as well as 2EXPTIME completeness of the TCL satisfiability problem for turn-based game graphs. TCL thus provides a better balance between expressiveness and complexity / efficiency considerations than ATL*, GL [1], and SL [7,15,14]. Given the expressive power as exemplified by the specifications from above, TCL can be viewed as an expressive yet inexpensive subclass of SL [15,14].

[1] A strategy is forgiving if it does not always punish betrayal in the previous round.

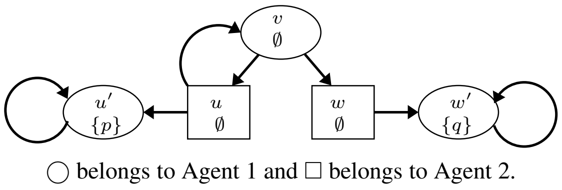


Fig. 1. A turn-based game graph

Organisation of the Paper. Section 2 explains turn-based game graphs for the description of multi-agent systems and presents the syntax and semantics of TCL. Section 3 discusses the expressiveness of TCL, establishing that CTL, ATL, LTL, and CTL* can be viewed as syntactic fragments of TCL. We show that TCL is more expressive than any of these logics while incomparable with ATL*, AMC, and GL [1] in expressiveness, and discuss the effect of a mild extension of TCL. In the following sections, we develop an automata based model-checking algorithm and establish the EXPTIME-completeness and 2EXPTIME-completeness of the TCL model-checking and satisfiability problem, respectively. Finally, we have implement a model-checker and validated the feasibility of using TCL on a set of benchmarks.

2 System Models and TCL

2.1 Turn-Based Game Graphs

A *turn-based* game is played by a finite number m of agents, indexed 1 through m. A game is a tuple $\mathcal{G} = \langle m, \mathcal{Q}, r, \omega, P, \lambda, E\rangle$, where

- Parameter m is the number of agents in the game.
- $\mathcal{Q}$ is the set of states and $r \in \mathcal{Q}$ is the *initial state* (or root) of $\mathcal{G}$.
- $\omega : \mathcal{Q} \mapsto [1, m]$ is a function that specifies the owner of each state. Only the owner of a state makes choices at the state.
- P is a finite set of atomic propositions.
- $\lambda : \mathcal{Q} \mapsto 2^P$ is a proposition labelling function.
- $E \subseteq \mathcal{Q} \times \mathcal{Q}$ is the set of transitions.

For ease of notation, we denote with $\mathcal{Q}_a = \{q \in \mathcal{Q} \mid \omega(q) = a\}$ the states owned by an agent a.

In Figure 1, we have the graphical representation of a turn-based game graph. The ovals and squares represent states while the arcs represent state transitions. We also put down the λ values inside the corresponding states.

For convenience, in the remaining part of the manuscript, we assume that we are always in the context of a given game graph $\mathcal{G} = \langle m, \mathcal{Q}, r, \omega, \mathcal{P}, \lambda, \mathcal{E}\rangle$. Thus, when we write $\mathcal{Q}, r, \omega, \mathcal{P}, \lambda$, and $\mathcal{E}$, we respectively refer to the components $\mathcal{Q}$, r, ω, P, λ, and E of this $\mathcal{G}$.

A *play* ρ is an infinite path $q_0 q_1 \ldots$ in $\mathcal{G}$ such that, for every $k \in \mathbb{N}$, $(q_k, q_{k+1}) \in \mathcal{E}$. ρ is *initial* if $q_0 = r$. For every $k \geq 0$, we let $\rho(k)$ denote q_k. Also, given $h \leq k$, we let $\rho[h, k]$ denote $\rho(h) \ldots \rho(k)$ and $\rho[h, \infty)$ denote the infinite tail of ρ from $\rho(h)$.

A *play prefix* is a finite segment of a play from the beginning of the play. Given a play prefix $\pi = q_0 q_1 \dots q_n$, $|\pi| = n+1$ denotes the length of the prefix. Given a $k \in [0, |\pi|-1]$, we let $\pi(k) = q_k$. For convenience, we use $\mathit{last}(\pi)$ to denote the last state in π, i.e., $\pi(|\pi|-1)$.

For an agent $a \in [1,m]$, a *strategy* σ for a is a function from $\mathcal{Q}^*\mathcal{Q}_a$ to $\mathcal{Q}$ such that for every $\pi \in \mathcal{Q}^*\mathcal{Q}_a$, $\sigma(\pi) \in \mathcal{Q}$ with $\big(\mathit{last}(\pi), \sigma(\pi)\big) \in \mathcal{E}$.

An *agency* A of $[1,m]$ is a subset of $[1,m]$. In a short hand notation, we often drop the curly brackets in the set notation, in particular for singleton and empty sets. For example, "$1,3,4$" is a short hand for $\{1,3,4\}$.

A play ρ is *compatible* with a strategy σ_a of an agent $a \in [1,m]$ iff, for every $k \in \mathbb{N}$, $\omega(\rho(k)) = a$ implies $\rho(k+1) = \sigma(\rho[0..k])$.

2.2 TCL Syntax

A TCL formula ϕ is constructed with the following three syntax rules.

$$\begin{aligned}
\phi &::= p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle A\rangle\psi \\
\psi &::= \phi \mid \eta \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \langle +A\rangle\psi_1 \mid \langle +A\rangle \bigcirc \psi_1 \mid \langle +A\rangle\eta_1 \mathrm{U}\psi_1 \mid \langle +A\rangle\psi_1 \mathrm{R}\eta_1 \\
&\quad \mid \ \langle -A\rangle\psi_1 \mid \langle -A\rangle \bigcirc \psi_1 \mid \langle -A\rangle\eta_1 \mathrm{U}\psi_1 \mid \langle -A\rangle\psi_1 \mathrm{R}\eta_1 \\
\eta &::= \phi \mid \eta_1 \vee \eta_2 \mid \eta_1 \wedge \eta_2 \mid \langle +\rangle \bigcirc \eta_1 \mid \langle +\rangle\eta_1 \mathrm{U}\eta_2 \mid \langle +\rangle\eta_1 \mathrm{R}\eta_2 \\
&\quad \mid \ \langle -A\rangle \bigcirc \eta_1 \mid \langle -A\rangle\eta_1 \mathrm{U}\eta_2 \mid \langle -A\rangle\eta_1 \mathrm{R}\eta_2
\end{aligned}$$

Here, p is an atomic proposition in $\mathcal{P}$ and $A \subseteq \{1,\dots,m\}$ is an agency. Property $\langle A\rangle\psi_1$ is an (existential) strategy quantification (SQ) specifying that there exist strategies of the agents in A that make all plays consistent with these strategies satisfy ψ_1. Property $\langle +A\rangle\psi_1$ is an (existential) strategy interaction quantification (SIQ) and can only occur bound by an SQ. Intuitively, $\langle +A\rangle\psi_1$ means that there exist strategies of the agents in A that work with the strategies introduced by the ancestor formulas. Likewise, $\langle -A\rangle$ indicates a revocation of the strategy binding for the agents in A. $\langle +\rangle$ is an abbreviation for $\langle +\emptyset\rangle$ or, equivalently $\langle -\emptyset\rangle$. Thus, it neither binds nor revokes the binding of the strategy of any agent. Yet, it provides a temporalisation in that it provides a tree formula that can be interpreted at a particular point.

'U' is the *until* operator. The property $\psi_1 \mathrm{U}\psi_2$ specifies a play along which ψ_1 is true until ψ_2 becomes true. Moreover, along the play, ψ_2 must eventually be fulfilled. 'R' is the *release* operator. Property $\psi_1 \mathrm{R}\psi_2$ specifies a play along which either ψ_2 is always true or $\psi_2 \mathrm{U}(\psi_1 \wedge \psi_2)$ is satisfied. (Release is dual to until: $\neg(\phi_1 \mathrm{U}\phi_2) \Leftrightarrow \neg\phi_2 \mathrm{R}\neg\phi_1$.)

In the following we may use $\langle ?A\rangle\psi$ to conveniently denote an SQ or SIQ formula with '?' is empty, '+', or '-'. An SIQ $\langle \pm A\rangle\psi$ is called non-trivial if A is not empty, and trivial otherwise.

Formulas ϕ are called *TCL formulas*, *sentences*, or *state formulas*. Formulas ψ and η are called *tree formulas*. Note that we strictly require that non-trivial strategy interaction cannot cross path modal operators. This restriction is important because it offers a sufficient level of locality to efficiently model-check a system against a TCL property. To illustrate this and to provide a simple extension that offers more expressive power to the cost of a much higher complexity, we informally discuss a small extension, *extended TCL* (ETCL), where the production rule of ψ also contains $\neg\psi$ and show that it can be used to encode ATL*, and the realisability problem of prenex QPTL can be reduced to ETCL model-checking.

For convenience, we also have the following shorthand notations.

$$\begin{aligned} \mathit{true} &\equiv p \vee (\neg p) & \mathit{false} &\equiv \neg \mathit{true} \\ \phi_1 \wedge \phi_2 &\equiv \neg((\neg\phi_1) \vee (\neg\phi_2)) & \phi_1 \Rightarrow \phi_2 &\equiv (\neg\phi_1) \vee \phi_2 \\ \Diamond\phi_1 &\equiv \mathit{true}\,\mathrm{U}\phi_1 & \Box\phi_1 &\equiv \mathit{false}\,\mathrm{R}\phi_1 \\ \neg \bigcirc \phi_1 &\equiv \bigcirc\neg\phi_1 & \langle A\rangle \bigcirc \psi_1 &\equiv \langle A\rangle\langle +\rangle \bigcirc \psi_1 \\ \langle A\rangle\psi_1\mathrm{U}\psi_2 &\equiv \langle A\rangle\langle +\rangle\psi_1\mathrm{U}\psi_2 & \langle A\rangle\psi_1\mathrm{R}\psi_2 &\equiv \langle A\rangle\langle +\rangle\psi_1\mathrm{R}\psi_2 \end{aligned}$$

In general, it would also be nice to have the universal SQs and SIQs as duals of existential SQs and SIQs, respectively. Couldn't we add, or encode by pushing negations to state formulas, a property of the form $[+A]\psi_1$, meaning that, for all strategies of agency A, ψ_1 will be fulfilled? In principle, this is indeed no problem, and extending the semantics would be simple. This logic would be equivalent to allowing for negations in the production rule of ψ. The problem with this logic is that it is too succinct. We will briefly discuss in the following section that model-checking becomes non-elementary if we allow for such negations.

From now on, we assume that we are always in the context of a given TCL sentence.

2.3 TCL Semantics

In order to prepare the definition of a semantics for TCL formulas, we start with the definition of a semantics for sentences of the form $\langle A\rangle\psi$, where ψ does not contain any SQs. We call these formulas *primitive* TCL formulas.

Due to the design of TCL, strategy bindings can only effectively happen at non-trivial SQs $\langle A\rangle$ and when a non-trivial SIQ $\langle +B\rangle$ is interpreted. To ease referring to these strategies, we first define the *bound agency* of a subformulas ϕ of a TCL sentence χ, denoted $\mathit{bnd}(\phi)$, as follows.

- For state formulas ϕ, $\mathit{bnd}(\phi) = \emptyset$.
- For state formulas $\langle A\rangle\psi$, $\mathit{bnd}(\psi) = A$ (unless ψ is a state formula).
- For tree formulas $\psi_1 = \langle +A\rangle\psi_2$, $\mathit{bnd}(\psi_2) = \mathit{bnd}(\psi_1) \cup A$.
- For tree formulas $\psi_1 = \langle -A\rangle\psi_2$, $\mathit{bnd}(\psi_2) = \mathit{bnd}(\psi_1) \smallsetminus A$.
- For all other tree formulas ψ_1 or ψ_2 with $\psi = \psi_1 \mathsf{OP} \psi_2$, with $\mathsf{OP} \in \{\wedge, \vee, \mathcal{U}, \mathcal{R}\}$, we have $\mathit{bnd}(\psi_1) = \mathit{bnd}(\psi)$ or $\mathit{bnd}(\psi_2) = \mathit{bnd}(\psi)$, respectively.

bnd shows, which agents have strategies assigned to them by an SIQ or SQ. Note that this leaves the *bnd* undefined for all state formulas not in the scope of an SQ formulas. For completeness, we could define *bnd* as empty in these cases, but a definition will not be required in the definition of the semantics.

As the introduction of additional strategies through non-trivial SIQ $\langle +B\rangle$ is governed by a *positive* Boolean combination, all strategy selections can be performed concurrently. Such a design leads us to the concept of strategy schemes.

A *strategy scheme* σ is the set of strategies introduced by any non-trivial SQ $\langle A\rangle$ or SIQ $\langle +A\rangle$. By abuse of notation, we use $\sigma[\phi, a]$ to identify such a strategy. Read in this way, σ can be viewed as a partial function from subformulas and their bound agencies to strategies. Thus, $\sigma[\phi, a]$ is defined if $a \in \mathit{bnd}(\phi)$ is in the bound agency of ϕ.

For example, given a strategy scheme σ for a TCL sentence $\langle 1\rangle\Diamond((\langle +2\rangle \bigcirc p) \wedge \langle 2\rangle\Box q)$, the strategy used in σ by Agent 1 to enforce the whole formula can be referred to by

$$\sigma[\langle 1\rangle\Diamond((\langle +2\rangle \bigcirc p) \wedge \langle 2\rangle\Box q), 1],$$

but also by $\sigma[\langle +2\rangle \bigcirc p, 1]$, while $\sigma[\langle 2\rangle\Box q, 1]$ is undefined.

We use a simple tree semantics for TCL formulas. A (computation) tree T_r is obtained by unravelling $\mathcal{G}$ from r and expand the ownership and labelling functions from $\mathcal{G}$ to T_r in the natural way. Technically, we have the following definition.

Definition: Computation Tree. A *computation tree* for a turn based game $\mathcal{G}$ from a state q, denoted T_q, is the smallest set of play prefixes that contains q and, for all $\pi \in T$ and $(\mathit{last}(\pi), q') \in \mathcal{E}$, $\pi q' \in T$. ■

The *strategy-pruned* tree for a tree node π, a strategy scheme σ, and a subformula ψ_1 of χ from a state q, in symbols $T_q\langle \pi, \sigma, \psi_1\rangle$, is the smallest subset of T_q such that:

- $\pi \in T_q\langle \pi, \sigma, \psi_1\rangle$;
- for all $\pi' \in T_q\langle \pi, \sigma, \psi_1\rangle$ with $\omega\big((\mathit{last}(\pi')\big) \notin \mathit{bnd}(\psi_1)$ and $(\mathit{last}(\pi'), q') \in \mathcal{E}$, $\pi' q' \in T_q\langle \pi, \sigma, \psi_1\rangle$;
- for all $\pi' \in T_q\langle \pi, \sigma, \psi_1\rangle$, $a = \omega\big((\mathit{last}(\pi')\big)$, and $q' = \sigma[\psi_1, a](\pi')$ with $a \in \mathit{bnd}(\psi_1)$, $\pi' q' \in T_q\langle \pi, \sigma, \psi_1\rangle$.

Given a computation tree or a strategy-pruned tree T and a node $\pi \in T$, for every $\pi q \in T$, we say that πq is a successor of π in T. A play ρ is a *limit of* T (or an infinite path in T), in symbols $\rho \overset{\infty}{\in} T$, if there are infinitely many prefixes of ρ in T.

We now define the semantics of subformulas of primitive TCL formulas inductively as follows. Given the computation tree T_q of $\mathcal{G}$, a tree node $\pi \in T_q$, and a strategy scheme σ, we write $T_q, \pi, \sigma \models \psi_1$ to denote that T_q satisfies ψ_1 at node π with strategy scheme σ.

While the notation might seem heavy on first glance, note that the truth for state formulas merely depends on the state $\mathit{last}(\pi)$ in which they are interpreted, and the tree formulas are simply interpreted on a strategy pruned tree rooted in π and defined by the strategy scheme.

- For state formulas ϕ other than SQ formulas, we use the state formula semantics: $T_q, \pi, \sigma \models \phi$ iff $\mathcal{G}, \mathit{last}(\pi) \models \phi$, with the usual definition.
  - $\mathcal{G}, q \models p$ if, and only if, $p \in \lambda(q)$,
  - $\mathcal{G}, q \models \neg\phi$ if, and only if, $\mathcal{G}, q \not\models \phi$,
  - $\mathcal{G}, q \models \phi_1 \vee \phi_2$ if, and only if, $\mathcal{G}, q \models \phi_1$ or $\mathcal{G}, q \models \phi_2$, and
  - $\mathcal{G}, q \models \phi_1 \wedge \phi_2$ if, and only if, $\mathcal{G}, q \models \phi_1$ and $\mathcal{G}, q \models \phi_2$.

  (Note that this allows for using negation for state formulas.)
- $T_q, \pi, \sigma \models \psi_1 \vee \psi_2$ iff $T_q, \pi, \sigma \models \psi_1$ or $T_q, \pi, \sigma \models \psi_2$. (The ψ_i are no state formulas.)
- $T_q, \pi, \sigma \models \psi_1 \wedge \psi_2$ iff $T_q, \pi, \sigma \models \psi_1$ and $T_q, \pi, \sigma \models \psi_2$ hold.
- $T_q, \pi, \sigma \models \langle \pm A\rangle \bigcirc \psi$ iff, for all successors $\pi q'$ of π in $T_q\langle \pi, \sigma, \langle \pm A\rangle \bigcirc \psi_1\rangle$, $T_q, \pi q', \sigma \models \psi$ holds.
- $T_q, \pi, \sigma \models \langle \pm A\rangle \psi_1 \mathrm{U} \psi_2$ iff, for all limits $\rho \overset{\infty}{\in} T_q\langle \pi, \sigma, \langle \pm A\rangle \psi_1 \mathrm{U} \psi_2\rangle$, there is a $k \geq |\pi| - 1$ such that $T_q, \rho[0,k], \sigma \models \psi_2$ and, for all $h \in [|\pi| - 1, k - 1]$, $T_q, \rho[0,h], \sigma \models \psi_1$ hold.

- $T_q, \pi, \sigma \models \langle \pm A \rangle \psi_1 \mathrm{R} \psi_2$ iff, for all limits $\rho \overset{\infty}{\in} T_q \langle \pi, \sigma, \langle \pm A \rangle \psi_1 \mathrm{R} \psi_2 \rangle$, one of the following two restrictions are satisfied.
  - For all $k \geq |\pi| - 1$, $T_q, \rho[0,k], \sigma \models \psi_2$.
  - There is a $k \geq |\pi| - 1$ such that $T_q, \rho[0,k], \sigma \models \psi_1 \wedge \psi_2$, and, for all $h \in [|\pi| - 1, k]$, $T_q, \rho[0,h], \sigma \models \psi_2$.
- $T_q, \pi, \sigma \models \langle \pm A \rangle \psi_1$ iff $T_q, \pi, \sigma \models \psi_1$.
- $\mathcal{G}, q \models \langle A \rangle \psi_1$ iff there is a strategy scheme σ such that $T_q, q, \sigma \models \psi_1$.

If ϕ_1 is a TCL sentence then we write $\mathcal{G} \models \phi_1$ for $\mathcal{G}, r \models \phi_1$.

Note that, while asking for the existence of a strategy scheme refers to all strategies introduced by some SQ or SIQ in the TCL sentence, only the strategies introduced by the respective SQ and the SIQs in its scope are relevant.

The simplicity of the semantics is owed to the fact that it suffices to introduce new strategies at the points where eventualities become true for the first time. Thus, they do not really depend on the position in which they are invoked and we can guess them up-front. (Or, similarly, together with the points on the unravelling where they are invoked.) This is possible, simply because the validity of state formulas (and hence of TCL sentences) cannot depend on the validity of the left hand side of an until (or the right hand side of a release) *after* the first time it has been satisfied.

3 Expressiveness of TCL

Note that TCL is not a superclass of BSIL since BSIL allows for negation in front of SIQs while TCL does not. However, by examining the proofs in [21] for the inexpressibility of BSIL properties by ATL*, GL, and AMC, we find that the BSIL sentence used in the proofs is also a TCL sentence. This leads to the conclusion that there are properties expressible in TCL but cannot be expressed in ATL*, GL, and AMC.

Lemma 1. *There are TCL sentences that cannot be expressed in any of ATL*, GL, or AMC.* ■

TCL is, in fact, not only a powerful logic, but also contains important logics either as syntactical fragments or can embed them in a straight forward way. ATL and CTL can be viewed as syntactic fragments of TCL.

But it is also simple to embed LTL and even CTL*. We start with $\exists$LTL, the less used variant where one is content if one path satisfies the formula. We then translate an LTL formula, which we assume w.l.o.g. to be in negative normal form (negations only in front of atomic propositions). Then "there is a path that satisfies ϕ" is equivalent to $\langle 1, \ldots, m \rangle \widehat{\phi}$, where $\widehat{\phi}$ is derived from ϕ by replacing every occurrence of $\bigcirc$, U, and R by $\langle + \rangle \bigcirc$, $\langle + \rangle$U, and $\langle + \rangle$R, respectively. The simple translation is possible because the formula $\widehat{\psi}$ is de-facto interpreted over a path, the path formed by the joint strategy of the agency $[1, m]$. The $\langle + \rangle$ operators we have added have no effect on the semantics in such a case, just as a CTL formula can be interpreted as the LTL formula obtained by deleting all path quantifiers when interpreted over a word.

Consequently, we have the expected semantics for $\forall LTL$: "all paths satisfy ϕ" is equivalent to $\neg \langle A \rangle \widehat{\neg \phi}$, where $\neg \phi$ is assumed to be re-written in negative normal form. The encoding of $\exists$LTL and $\forall$LTL can easily be extended to the encoding of CTL*.

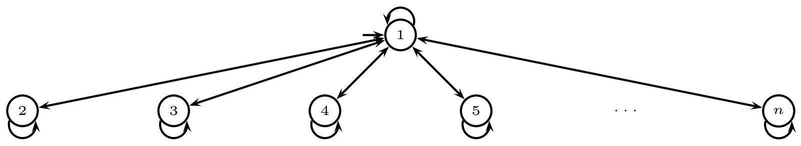


Fig. 2. The turn-based game graph from the non-elementary hardness proof of extended TCL

Lemma 2. *TCL is more expressive than CTL* and LTL.* ■

This encoding does not extend to ATL*. $\langle 1\rangle((\Box p)\vee\Box q)$ is an ATL* property that cannot be expressed with TCL.

This is different from the ATL property $(\langle 1\rangle\Box p)\vee\langle 1\rangle\Box q$ or the TCL property $\langle 1\rangle((\langle +\rangle\Box p)\vee\langle +\rangle\Box q)$. In fact, the proofs and examples in [21] can also be applied in this work to show that there are properties of ATL* (or GL, or AMC) that cannot be expressed with TCL. This leads to the following lemma.

Lemma 3. *TCL is incomparable in expressiveness with ATL*, GL, and AMC.* ■

Note, however, that allowing for a negation in the definition of ψ would change the situation. Then an ATL* formula $\langle A\rangle\psi$ (assuming for the sake of simplicity that ψ is an LTL formula), would become $\langle A\rangle\neg\langle +[1,m]\smallsetminus A\rangle\widehat{\neg\psi}$ in the extended version of TCL. The translation extends to full ATL*, but this example also demonstrates why negation is banned: even without nesting, we can, by encoding ATL*, encode a 2EXPTIME complete model-checking problem, losing the appealing tractability of our logic.

In fact, it is easy to reduce the realisability problem of prenex QPTL, and hence a non-elementary problem, to the model-checking problem of extended TCL. Using the game structure from Figure 2, we can encode the realisability of a prenex QPTL formula with $n-1$ variables, for simplicity of the form $\forall p_2\exists p_3\forall p_4\ldots\exists p_n\phi$, where $p_2,\ldots,p_n$ are all propositions occurring in ϕ. We reduce this to model-checking the formula

$$\phi' = \langle 1\rangle\neg\langle +2\rangle\neg\langle +3\rangle\neg\langle +4\rangle\neg\ldots\neg\langle +n\rangle(\psi_\phi\wedge\langle +\rangle\Box p_1),$$

where ψ_ϕ can be obtained from $\widehat{\phi}$ by replacing

- every literal p_i by $\langle -1\rangle\langle +1\rangle\bigcirc(p_i\wedge\langle +\rangle\bigcirc p_i)$, and
- every literal $\neg p_i$ by $\langle -1\rangle\langle +1\rangle\bigcirc(p_i\wedge\langle +\rangle\bigcirc\neg p_i)$.

These formulas are technically not extended TCL formulas as $\langle +i\rangle\psi_1$ is not part of the production rule of ψ, but $\langle +i\rangle\psi_1$ can be used as an abbreviation for $\langle +i\rangle$*false*$\mathrm{U}\psi_1$.

Checking satisfiability of ϕ is is equivalent to model-checking ϕ' on the game shown in Figure 2. The game has $n+1$ nodes, agents, and atomic propositions. The nodes in Figure 2 are labeled with the agent that owned the nodes, and the atomic proposition p_i is true exactly in node i. From his state, Agent 1 can move to any other state, while all other agents can either stay in their state or return to the state owned by Agent 1.

The game starts in the node owned by Agent 1, and in order to comply with the specification, the outermost strategy profile chosen by Agent 1 must be to stay in the initial state for ever. ψ_ϕ is chosen to align the truth of p_i at position $j\in\mathbb{N}$ with the decision that Agent i makes on the history $1^j i$: *true* corresponds to staying in i and *false* with returning to 1.

It is not hard to establish a matching upper bound for model-checking extended TCL.

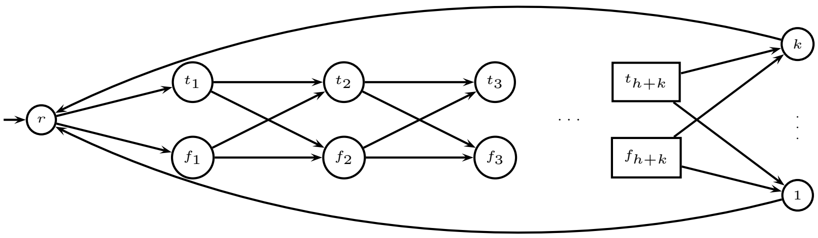


Fig. 3. The turn-based game graph from the EXPTIME hardness proof

4 Complexity of TCL

In this section, we show that model-checking TCL formulas is EXPTIME-complete in the formula and P-complete in the model (and for fixed formulas), while the satisfiability problem is 2EXPTIME-complete. As the proof of inclusion of the satisfiability problem in 2EXPTIME builds on the proof of the inclusion of model-checking in EXPTIME, we start with an outline of the EXPTIME hardness argument for the TCL model-checking problem and then continue with describing EXPTIME and 2EXPTIME decision procedures for the TCL model and satisfiability checking problem, respectively. 2EXPTIME hardness for TCL satisfiability is implied by the inclusion of CTL* as a de-facto sub-language [20].

We show EXPTIME hardness by a reduction from the PEEK-G_6 [19] game. An instance of PEEK-G_6 consists of two disjoint sets of boolean variables, $P_1 = \{p_1, \ldots, p_h\}$ (owned by a safety agent) and $P_2 = \{p_{h+1}, \ldots, p_{h+k}\}$ (owned by a reachability agent), a subset $I \subseteq P_1 \cup P_2$ of them that are initially *true*, and a boolean formula γ in CNF over $P_1 \cup P_2$ that the reachability agent wants to become *true* eventually. The game is played in turns between the safety and the reachability agent (say, with the safety agent moving first), and each player can change the truth value of one of his or her variables in his/her turn.

Lemma 4. *TCL model-checking is EXPTIME hard for primitive TCL formulas.*

Proof. To reduce determining the winner of an instance of a PEEK-G_6 game to TCL model-checking, we introduce a 2-agent game $\mathcal{G} = \langle 2, \mathcal{Q}, r, \omega, \mathcal{P}, \lambda, \mathcal{E} \rangle$ as shown in Figure 3, where Agent 1 (he, for convenience) represents the safety agent while Agent 2 (she, for convenience) represents the reachability agent. t_{h+k} and f_{h+k} are the only states owned by Agent 2.

The game is played in rounds, and a round starts each time the game is at state r. If the game goes through t_i this is identified with the variable p_i to be true. Likewise, going through f_i is identified with the variable being false.

It is simple to write a TCL specification that forces the safety player to toggle the value of exactly one of his variables in each round, and to toggle the value of the variable p_{h+i} of the reachability player defined by the state i she has previously moved to, while maintaining all other variable values. Requiring additionally that the safety agent can guarantee that the boolean formula is never satisfied provides the reduction. ■

The details of the construction are available in the full version. It is interesting that a game with only two agents suffices for the proof. Two agents are also sufficient to show P hardness for fixed formulas, as solving a reachability problem for AND-OR graphs [12] naturally reduces to showing $\langle 1 \rangle \Diamond p$.

Lemma 5. *TCL model-checking for fixed formulas is P hard for primitive TCL formulas.* ∎

In order to establish inclusion in EXPTIME and P, respectively, we use an automata based argument.

Theorem 1. *The model-checking problem of TCL formulas against turn-based game graphs is EXPTIME-complete, and P-complete for fixed formulas.*

Proof. We first show the claim for primitive TCL formulas $\phi = \langle A \rangle \psi$.

To keep the proof simple, we first consider a tree automaton $\mathcal{U}$ that checks the acceptance of ψ for a given strategy scheme σ. That is, $\mathcal{U}$ checks if ${T_q}^+, q, \sigma \models \psi$ under the assumption that both σ and the truth values for the subformulas starting with a $\langle \pm B \rangle$ are encoded in the nodes of ${T_q}^+$.

Such an automaton would merely have to run simple consistency checks, and it is simple to construct a suitable universal weak tree automaton $\mathcal{U}$, which is polynomial in the size of ϕ. From there it is simple to infer a deterministic Büchi tree automaton $\mathcal{D}$, which is exponential in the weak universal tree automaton [16].

It is then a trivial step (projection) to *guess* σ and the truth annotation of the subformulas on the fly, turning the deterministic Büchi tree automaton $\mathcal{D}$ that requires a correct annotation into a nondeterministic Büchi automaton $\mathcal{N}$ of the same size that checks $\mathcal{G}, q \models \phi$. Acceptance can be checked in time quadratic in the size of the product of $\mathcal{N}$ and $\mathcal{G}$ [6].

To take the step to full TCL, we can model-check the truth of primitive TCL formulas and then use the result of this model-checking instead of the respective subformula.

Hardness is inherited from Lemmata 4 and 5. ∎

This argument shows more: the complexity of TCL model-checking for fixed formulas does not depend on the formula. It suffices to solve a number of Büchi games, where both the size of the game and the number of games to be played is linear in $\mathcal{G}$.

Corollary 1. *Viewing the size of a TCL sentence as a parameter, TCL model-checking is fixed parameter tractable.*

The automata construction from the proof of Theorem 1 extends to a construction for satisfiability checking.

Theorem 2. *The TCL satisfiability problem is 2EXPTIME-complete.*

Proof. As usual, it is convenient to construct an enriched model that contains the truth of all subformulas for a TCL sentence ϕ that start with an SQ.

In a first step, we construct an alternating tree automaton $\mathcal{A}$ that recognises the enriched models of a specification. This is quite simple: $\mathcal{A}$ merely has to check that the boolean combination of SQ formulas that forms the TCL sentence ϕ is satisfied and that the truth assignment of each SQ is consistent. But this is simple, as we can use the tree automaton $\mathcal{N}_{\phi'}$ from the proof for Theorem 1 to validate the claim that a subformula ϕ' of ϕ that starts with an SQ is true, and its dual to validate that it is false. Hence, such an automaton has only two states more than the sum of the states of the individual $\mathcal{N}_{\phi'}$. In particular, it is exponential in ϕ.

For the resulting alternating automaton, we can again invoke the simulation theorem [16] to construct an equivalent nondeterministic parity automaton, which has doubly exponentially many states in ϕ (and whose transition table is doubly exponential in ϕ) and whose colours are exponential in ψ. Solving the emptiness game of this automaton reduces to solving a parity game, which can be done in time doubly exponential in ψ, e.g., using [18].

Hardness is inherited from CTL* satisfiability checking [20]. ■

5 Implementation and Experiment

As a proof of concept, we have implemented a model-checker, `tcl`, in C++. `tcl` accepts models composed of extended automata that communicate with synchronisers and shared variables, with an explicit shared variable `turn` that specifies the turn of agents at a state. A turn-based game graph is then constructed as the product of the extended automata. Such an input format facilitates modular description of the interaction among the agents.

The implementation builds on a prototype for a PSPACE logic [21]. The extension is possible because we can reduce the complexity of TCL to PSPACE by simply restricting the number of operators in the η production rules in the scope of any SQ to be logarithmic in the size of the TCL sentence. We show this for primitive TCL sentences.

Lemma 6. *Model-checking can be done in space bilinear in the size of the turn based game structure and the state and tree formulas that are produced using the ψ production rules and exponentially only in the number of η produced tree formulas.*

Proof. We have seen that, for a primitive TCL sentence ϕ, we can use a single strategy scheme and only have to refer to the *first* position that the right hand side of an until or the left hand side of a release operator is true. Moreover, it suffices to guess just a minimal set of positions where tree formulas are true. In particular, the left hand side of a release, the right hand side of an until, and a next formula are then marked true exactly once, and the respective release and until formulas never need to be marked as true after such an event.

We can therefore use an alternating algorithm that guesses such minimal truth claims. The algorithm alternates between a verifier who guesses a truth assignment and the current decisions of the strategy scheme, and a falsifier, who guesses the direction into which to expand the path.

It is now easy to see that they will produce an infinite path in this way, and on this path each obligation that refers to a tree subformula from a ψ production rule can appear only on a continuous interval. The points where these obligations change is therefore linear in the size of ϕ. However, it also needs to track the truth value of tree formulas produced by the η production rule. (If there are multiple untilities introduced by η production rules, this also includes a marker that distinguishes a leading until, which is changed in a round robin fashion when the leading untility is fulfilled.)

The number of possible assignments is then exponential in the number of tree subformulas from η production rules. Note that $\Box$ formulas can be exempt from this rule: they are monotonous and hence incur a small impact similar to the formulas introduced using the ψ production rule.

Hence, if $|\mathcal{G}|$ denotes the size of the turn based game and k the number of temporal operators (different to $\Box$) introduced by η production rules, we end up in a cycle if there is no change in the truth assignment temporal operators that are introduced by ψ production rules or $\Box$ operators we reach a cycle within $|\mathcal{G}| \cdot k \cdot 2^k$ steps. Hence, we reach a cycle in a number of steps that is linear in $|G|$ and the size of ϕ, and exponential only in the size of η-produced temporal operators (different to $\Box$).

Upon reaching a cycle, is suffices to check if the cycle is accepting. (No standing obligation by an until.) ■

The model-checker uses a stack to explicitly enumerate all paths of all tree tops with depth prescribed by Lemma 6. The tool can be downloaded from Sourceforge at project REDLIB at: `http://sourceforge.net/projects/redlib/`.

We use the parametrised models of the iterated prisoners' dilemma as our benchmarks to check the performance of our implementation. A brief explanation of the models can be found in the introduction. The unique parameter to the models are the number of prisoners m. There is also a policeman in the models. We build a turn-based game graph for each value of m in the experiments. The parametrisation helps us to observe how our algorithm and implementation scale to model and formula sizes. To simplify the construction of the state-space representation, we assume that, in each iteration, the prisoners make their decisions in a fixed order. After all prisoners have made their decisions, the policeman makes his decision. Subsequently, the whole game moves to the next iteration. We use seven benchmark formulas on these models in our experiments. The first five benchmarks are taken from the examples (A) through (E) from the introduction. Benchmarks (F) and (G) are the following two properties, taken from [21].

- Property (F) specifies that all prisoners except Prisoner 1 can collaborate to release Prisoner 1 and let Prisoner 1 decide their fate.
  $\langle 2,\ldots,m\rangle\big((\langle +\rangle\Diamond\neg\mathtt{jail}_1) \wedge \bigwedge_{i\in\{2,\ldots m\}}(\langle +1\rangle\Diamond\neg\mathtt{jail}_i) \wedge (\langle +1\rangle\Box\mathtt{jail}_i\big)$ (F)
- Property (G) specifies that Prisoner 1 has a strategy to put all other prisoners in jail while leaving her fate to them.
  $\langle 1\rangle\big((\bigwedge_{i\in\{2,\ldots m\}}\langle +\rangle\Box\mathtt{jail}_i)\wedge(\langle 2,\ldots,m\rangle\Diamond\neg\mathtt{jail}_1)\wedge\langle 2,\ldots,m\rangle\Box\mathtt{jail}_1\big)$ (G)

For these benchmarks, we have collected the performance data for various parameter values in Table 1. For small models, the memory usage is dominated by the normal overhead, such as the representation of variable tables, state-transition tables, formula structures, etc. The data shows that our prototype can handle the various benchmarks, and scales well on five of the seven benchmarks. Ignoring the overhead, it also shows the exponential growth. The models, however, are growing exponentially, too. We assume that this growth i the main cause of the exponential growth of the response time.

Table 1. Performance data of model-checking the TCL fragment

properties \ m	2	3	4	5	6	7	8	9	10
(A)	0.71s	0.94s	5.41s	66.3s	945s	>1000s			
	163M	165M	185M	350M	1307M				
(B)	0.50s	0.52s	0.61s	0.71s	1.11s	1.62s	5.77s	20.9s	68.1s
	163M	163M	164M	165M	168M	176M	214M	270M	376M
(C)	0.51s	0.51s	0.6s	0.82s	1.01s	1.81s	5.54s	18.2s	48.3s
	163M	163M	164M	165M	168M	176M	200M	241M	318M
(D)	0.5s	0.51s	0.57s	0.74s	1.01s	1.79s	7.41s	33.8s	141s
	163M	163M	164M	165M	168M	175M	232M	312M	430M
(E)	0.51s	0.66s	19.1s	>1000s					
	163M	164M	194M						
(F)	0.51s	0.53s	0.61s	0.71s	1.01s	1.70s	5.38s	15.2s	53.7s
	163M	163M	163M	165M	168M	175M	202M	243M	295M
(G)	0.52s	0.52s	0.65s	0.72s	1.03s	1.85s	4.86s	16.1s	93.5s
	163M	163M	164M	165M	169M	177M	189M	208M	235M

s: seconds; M: megabytes.

The models are with 1 policeman and m prisoners. The experiment was carried out on an Intel i5 2.4G notebook with 2 cores and 4G memory, running ubuntu Linux version 11.10.

6 Conclusion

TCL is a promising logic for the specification of groups of agents who balance their strategies in order to cooperate with different partners to achieve different objectives. It is an inexpensive logic in many ways. First and foremost, it is fixed parameter tractable. Following folklore, specifications are tiny while models are huge. In this situation, fixed parameter tractability is a very important property, in particular as it is achieved by a natural and simple decision procedure, which is merely exponential in the formula.

This appealing property is not bought with inexpressiveness. In particular, the popular temporal logics LTL, CTL, ATL, and CTL* are contained as de-facto sublogics. Consequently, it can be excellently used to extend existing specifications in these languages, without the need to develop competitive models.

The applicability is underlined by compelling data from our benchmarks. This is in spite of the fact that our implementation is rather based on an ad hoc extension of an existing algorithm for a different logic, and neither fully exploit the low complexity, nor is a fully symbolic implementation. It will be interesting to see by which extent symbolic representation like BDDs will enhance the performance and how an automata based tool would fare.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM (JACM) 49(5), 672–713 (2002)
2. Axelrod, R.: Effective choice in the prisoner's dilemma. Journal of Conflict Resolution 24(1), 3–25 (1980)

3. Baier, C., Brázdil, T., Gröser, M., Kucera, A.: Stochastic game logic. In: QEST, pp. 227–236. IEEE Computer Society (2007)
4. Büchi, J., Landweber, L.: Definability in th emonadic second-order theory of successor. Journal of Symbolic Logic 34(2), 166–170 (1969)
5. Büchi, J., Landweber, L.: Solving sequential conditions by finite-state strategies. Trans. AMS 138(4), 295–311 (1969)
6. Chatterjee, K., Henzinger, M.: An $O(n^2)$ time algorithm for alternating Büchi games. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012), Kyoto, Japan, January 17-19, pp. 1386–1399. SIAM (2012)
7. Chatterjee, K., Henzinger, T.A., Piterman, N.: Strategy logic. Information and Computation 208, 677–693 (2010)
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. Costa, A.D., Laroussinie, F., Markey, N.: Atl with strategy contexts: Expressiveness and model checking. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010). Leibniz International Proceedings in Informatics (LIPIcs), vol. 8, pp. 120–132. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010)
10. Finkbeiner, B., Schewe, S.: Coordination Logic. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 305–319. Springer, Heidelberg (2010)
11. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. 23(5) (1997)
12. Immerman, N.: Number of quantifiers is better than number of tape cells. Journal of Computer and System Sciences 22(3), 65–72 (1981)
13. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of ACM 47(2), 312–360 (2000)
14. Mogavero, F., Murano, A., Perelli, G., Vardi, M.Y.: What Makes ATL* Decidable? A Decidable Fragment of Strategy Logic. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 193–208. Springer, Heidelberg (2012)
15. Mogavero, F., Murano, A., Vardi, M.Y.: Reasoning about strategies. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010). LIPIcs, vol. 8, pp. 133–144 (2010)
16. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin, McNaughton and Safra. Theoretical Computer Science 141(1-2), 69–107 (1995)
17. Pnueli, A.: The temporal logic of programs. In: 18th Annual IEEE-CS Symposium on Foundations of Computer Science, pp. 45–57 (1977)
18. Schewe, S.: Solving Parity Games in Big Steps. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 449–460. Springer, Heidelberg (2007)
19. Stockmeyer, L.J., Chandra, A.K.: Provably difficult combinatorial games. SIAM Journal on Computing (SICOMP) 8(2), 151–174 (1979)
20. Vardi, M., Stockmeyer, L.: Improved upper and lower bounds for modal logics of programs: Preliminary report. In: Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC 1985), Providence, Rhode Island, USA, May 6-8, pp. 240–251 (1985)
21. Wang, F., Huang, C.-H., Yu, F.: A Temporal Logic for the Interaction of Strategies. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 466–481. Springer, Heidelberg (2011)
22. Wilke, T.: Alternating tree automata, parity games, and modal μ-calculus. Bulletin of the Belgian Mathematical Society 8(2) (May 2001)

Synthesis from LTL Specifications with Mean-Payoff Objectives

Aaron Bohy[1], Véronique Bruyère[1], Emmanuel Filiot[2], and Jean-François Raskin[3,⋆]

[1] Université de Mons
[2] Université Paris-Est Créteil
[3] Université Libre de Bruxelles

Abstract. The classical LTL synthesis problem is purely qualitative: the given LTL specification is realized or not by a reactive system. LTL is not expressive enough to formalize the correctness of reactive systems with respect to some quantitative aspects. This paper extends the *qualitative* LTL synthesis setting to a *quantitative* setting. The alphabet of actions is extended with a weight function ranging over the integer numbers. The value of an infinite word is the mean-payoff of the weights of its letters. The synthesis problem then amounts to automatically construct (if possible) a reactive system whose executions all satisfy a given LTL formula and have mean-payoff values greater than or equal to some given threshold. The latter problem is called $\mathsf{LTL}_{\mathsf{MP}}$ synthesis and the $\mathsf{LTL}_{\mathsf{MP}}$ realizability problem asks to check whether such a system exists. By reduction to two-player mean-payoff parity games, we first show that $\mathsf{LTL}_{\mathsf{MP}}$ realizability is not more difficult than LTL realizability: it is 2ExpTime-Complete. While infinite memory strategies are required to realize $\mathsf{LTL}_{\mathsf{MP}}$ specifications in general, we show that ϵ-optimality can be obtained with finite-memory strategies, for any $\epsilon > 0$. To obtain efficient algorithms in practice, we define a Safraless procedure to decide whether there exists a finite-memory strategy that realizes a given specification for some given threshold. This procedure is based on a reduction to two-player energy safety games which are in turn reduced to safety games. Finally, we show that those safety games can be solved efficiently by exploiting the structure of their state spaces and by using antichains as a symbolic data-structure. All our results extend to multi-dimensional weights. We have implemented an antichain-based procedure and we report on some promising experimental results.

1 Introduction

Formal specifications of reactive systems are usually expressed using formalisms like the linear temporal logic (LTL), the branching time temporal logic (CTL), or automata formalisms like Büchi automata. Those formalisms allow one to express *Boolean properties* in the sense that a reactive system either conforms to them, or violates them. Additionally to those qualitative formalisms, there is a clear need for another family of formalisms that are able to express *quantitative properties* of reactive systems. Abstractly, a quantitative property can be seen as a function that maps an execution of a reactive system to a numerical value. For example, in a client-server application, this

⋆ Author supported by ERC Starting Grant (279499: inVEST).

numerical value could be the mean number of steps that separate the time at which a request has been emitted by a client and the time at which this request has been granted by the server along an execution. Quantitative properties are concerned with a large variety of aspects like quality of service, bandwidth, energy consumption,... But quantities are also useful to compare the merits of alternative solutions, e.g. we may prefer a solution in which the quality of service is high and the energy consumption is low. Currently, there is a large effort of the research community with the objective to lift the theory of formal verification and synthesis from the *qualitative world* to the richer *quantitative world* [15] (see related works for more details). In this paper, we consider mean-payoff and energy objectives. The alphabet of actions is extended with a weight function ranging over the integer numbers. A mean-payoff objective is a set of infinite words such that the mean value of the weights of their letters is greater than or equal to a given rational threshold [22], while an energy objective is parameterized by a non-negative initial energy level c_0 and contains all the words whose finite prefixes have a sum of weights greater than or equal to $-c_0$ [5].

In this paper, we participate to this research effort by providing theoretical complexity results, practical algorithmic solutions, and a tool for the automatic synthesis of reactive systems from *quantitative specifications* expressed in the linear time temporal logic LTL extended with (multi-dimensional) mean-payoff and (multi-dimensional) energy objectives. To illustrate our contributions, let us consider the following specification of a controller that should grant exclusive access to a resource to two clients.

Example 1. A client requests access to the resource by setting to true its request signal (r_1 for client 1 and r_2 for client 2), and the server grants those requests by setting to true the respective grant signal g_1 or g_2. We want to synthetize a server that eventually grants any client request, and that only grants one request at a time. This can be formalized in LTL as the conjunction of the three following formulas, where the signals in $I = \{r_1, r_2\}$ are controlled by the environment (the two clients), and the signals in $O = \{g_1, w_1, g_2, w_2\}$ are controlled by the server:

$$\phi_1 = \Box(r_1 \rightarrow \mathsf{X}(w_1 \mathsf{U} g_1)) \quad \phi_2 = \Box(r_2 \rightarrow \mathsf{X}(w_2 \mathsf{U} g_2)) \quad \phi_3 = \Box(\neg g_1 \vee \neg g_2)$$

Intuitively, ϕ_1 (resp. ϕ_2) specifies that any request of client 1 (resp. client 2) must be eventually granted, and in-between the waiting signal w_1 (resp. w_2) must be high. Formula ϕ_3 stands for mutual exclusion. Let $\phi = \phi_1 \wedge \phi_2 \wedge \phi_3$.

The formula ϕ is realizable. One possible strategy for the server is to alternatively assert w_2, g_1 and w_1, g_2, i.e. alternatively grant client 1 and client 2. While this strategy is formally correct, as it realizes the formula ϕ against all possible behaviors of the clients, it may not be the one that we expect. Indeed, we may prefer a solution that does not make unsollicited grants for example. Or, we may prefer a solution that gives, in case of request by both clients, some priority to client 2's request. In the later case, one elegant solution would be to associate a cost equal to 2 when w_2 is true and a cost equal to 1 when w_1 is true. This clearly will favor solutions that give priority to requests from client 2 over requests from client 1. We will develop other examples in the paper and describe the solutions that we obtain automatically with our algorithms.

Contributions – We now detail our contributions and give some hints about the proofs. In Section 2, we define the realizability problems for $\mathsf{LTL_{MP}}$ (LTL extended with mean-payoff objectives) and $\mathsf{LTL_E}$ (LTL extended with energy objectives). In Section 3, we show that, as for the LTL realizability problem, both the $\mathsf{LTL_{MP}}$ and $\mathsf{LTL_E}$ realizability problems are 2ExpTime-Complete. As the proof of those three results follow a similar structure, let us briefly recall how the 2ExpTime upper bound of the classical LTL realizability problem is established in [19]. The formula is turned into an equivalent nondeterministic Büchi automaton, which is then transformed into a deterministic parity automaton using Safra's construction. The latter automaton can be seen as a two-player parity game in which Player 1 wins if and only if the formula is realizable. For the $\mathsf{LTL_{MP}}$ (resp. $\mathsf{LTL_E}$) realizability problem, our construction follows the same structure, except that we go to a two-player parity game with an additional mean-payoff (resp. energy) objective. By a careful analysis of these two constructions, we build, on the basis of results in [8,11], solutions that provide the announced 2ExpTime upper bound.

Winning mean-payoff parity games may require infinite memory strategies, but there exist ϵ-optimal finite-memory strategies [11]. In contrast, for energy parity games, finite-memory optimal strategies always exist [8]. Those results transfer to $\mathsf{LTL_{MP}}$ (resp. $\mathsf{LTL_E}$) realizability problems thanks to their reduction to mean-payoff (resp. energy) parity games. Furthermore, we show that under finite-memory strategies, $\mathsf{LTL_{MP}}$ realizability is in fact equivalent to $\mathsf{LTL_E}$ realizability: a specification is MP-realizable under finite-memory strategies if and only if it is E-realizable, by simply shifting the weights of the signals by the threshold value. As finite-memory strategies are more interesting in practice, we thus concentrate on the $\mathsf{LTL_E}$ realizability problem in the rest of the paper.

Even if recent progresses have been made [21], Safra's construction is intricate and notoriously difficult to implement efficiently [1]. We develop in Section 4, following [17], a Safraless procedure for the $\mathsf{LTL_E}$ realizability problem, that is based on a reduction to a safety game, with the nice property to transform a quantitative objective into a simple qualitative objective. The main steps are as follows. *(1)* Instead of transforming an LTL formula into a deterministic parity automaton, we use a universal co-Büchi automaton as proposed in [17]. To deal with the energy objectives, we thus transform the formula into a universal co-Büchi energy automaton for some initial credit c_0, which requires that all runs on an input word w visit finitely many accepting states and the energy level of w is always positive starting from the credit c_0. *(2)* By strenghtening the co-Büchi condition into a K-co-Büchi condition as done in [20,14], where at most K accepting states can be visited by each run, we then go to an energy safety game. We show that for large enough value K and initial credit c_0, this reduction is complete. *(3)* Any energy safety game is equivalent to a safety game, as shown in [7].

In Section 5, our results are extended to the multi-dimensional case, i.e. tuples of weights. Finally, we discuss some implementation issues in Section 6. Our Safraless construction has two main advantages. *(1)* The search for winning strategies for $\mathsf{LTL_E}$ realizability can be incremental on K and c_0 (avoiding in practice the large theoretical bounds ensuring completeness). *(2)* The state space of the safety game can be partially ordered and solved by a backward fixpoint algorithm. Since the latter manipulates sets of states closed for this order, it can be made efficient and symbolic by working only on the antichain of their maximal elements. All the algorithms are implemented in our tool Acacia+ [3], and promising experimental results are reported in Section 6.

Due to lack of space, some proofs are omitted or just sketched. The full version is available at http://arxiv.org/abs/1210.3539.

Related Work – The LTL synthesis problem has been first solved in [19], Safraless approaches have been proposed in [16,17,20,14], and implemented in prototypes of tools [16,14,13,3]. All those works only treat plain qualitative LTL, and not the quantitative extensions considered in this article.

Mean-payoff games [22] and energy games [5,7], extensions with parity conditions [11,8,6], or multi-dimensions [10,12] have recently received a large attention from the research community. The use of such game formalisms has been advocated in [2] for specifying quantitative properties of reactive systems. Several among the motivations developed in [2] are similar that our motivations for considering quantitative extensions of LTL. All these related works make the assumption that the game graph is given explicitly, and not implicitly using an LTL formula, as in our case.

In [4], Boker et al. introduce extensions of linear and branching time temporal logics with operators to express constraints on values accumulated along the paths of a weighted Kripke structure. One of their extensions is similar to $\mathsf{LTL}_{\mathsf{MP}}$. However the authors of [4] only study the complexity of model-checking problems whereas we consider realizability and synthesis problems.

2 Problem Statement

Linear Temporal Logic – The formulas of linear temporal logic (LTL) are defined over a finite set P of atomic propositions. The syntax is given by the grammar:

$$\phi ::= p \mid \phi \vee \phi \mid \neg\phi \mid \mathsf{X}\phi \mid \phi \mathsf{U} \phi \qquad p \in P$$

LTL formulas ϕ are interpreted on infinite words $u \in (2^P)^\omega$ via a satisfaction relation $u \models \phi$ defined as usual [18]. Given ϕ, we let $[\![\phi]\!] = \{u \in (2^P)^\omega \mid u \models \phi\}$.

LTL *Realizability and Synthesis –* The realizability problem for LTL is best seen as a game between two players. Let ϕ be an LTL formula over the set $P = I \uplus O$ partitioned into I the set of *input signals* controlled by Player I (the environment), and O the set of *output signals* controlled by Player O (the controller). With this partition of P, we associate the three following alphabets: $\Sigma_P = 2^P$, $\Sigma_O = 2^O$, and $\Sigma_I = 2^I$. The realizability game is played in turns. Player O starts by giving $o_0 \in \Sigma_O$, Player I responds by giving $i_0 \in \Sigma_I$, then Player O gives $o_1 \in \Sigma_O$ and Player I responds by $i_1 \in \Sigma_I$, and so on. This game lasts forever and the outcome of the game is the infinite word $(o_0 \cup i_0)(o_1 \cup i_1)(o_2 \cup i_2) \cdots \in \Sigma_P^\omega$.

The players play according to *strategies*. A strategy for Player O is a mapping $\lambda_O : (\Sigma_O \Sigma_I)^* \to \Sigma_O$, while a strategy for Player I is a mapping $\lambda_I : (\Sigma_O \Sigma_I)^* \Sigma_O \to \Sigma_I$. The outcome of the strategies λ_O and λ_I is the word $\mathsf{Outcome}(\lambda_O, \lambda_I) = (o_0 \cup i_0)(o_1 \cup i_1) \dots$ such that $o_0 = \lambda_O(\epsilon)$, $i_0 = \lambda_I(o_0)$ and for $k \geq 1$, $o_k = \lambda_O(o_0 i_0 \dots o_{k-1} i_{k-1})$ and $i_k = \lambda_I(o_0 i_0 \dots o_{k-1} i_{k-1} o_k)$. We denote by $\mathsf{Outcome}(\lambda_O)$ the set of all outcomes

$\mathsf{Outcome}(\lambda_O, \lambda_I)$ with λ_I any strategy of Player I. We let Π_O (resp. Π_I) be the set of strategies for Player O (resp. Player I).

Given an LTL formula ϕ (the specification), the LTL *realizability problem* is to decide whether there exists $\lambda_O \in \Pi_O$ such that for all $\lambda_I \in \Pi_I$, $\mathsf{Outcome}(\lambda_O, \lambda_I) \models \phi$. If such a *winning* strategy exists, we say that the specification ϕ is *realizable*. The LTL *synthesis problem* asks to produce a strategy λ_O that realizes ϕ, when it is realizable.

Moore Machines – It is known that LTL realizability is 2ExpTime-Complete and that finite-memory strategies suffice to witness realizability [19]. A strategy $\lambda_O \in \Pi_O$ is *finite-memory* if there exists a right-congruence $\sim$ on $(\Sigma_O \Sigma_I)^*$ of finite index such that $\lambda_O(u) = \lambda_O(u')$ for all $u \sim u'$. It is equivalent to say that it can be described by a *Moore machine* $\mathcal{M}$, i.e. a finite deterministic state machine with output [19]. If the machine $\mathcal{M}$ describes λ_O, then $\mathsf{Outcome}(\lambda_O)$ is called the *language of* $\mathcal{M}$, denoted by $L(\mathcal{M})$. The memory size of the strategy is the index of $\sim$.

Theorem 1 ([19]). *The* LTL *realizability problem is 2ExpTime-Complete and any realizable formula is realizable by a finite-memory strategy with memory size* $2^{2^{O(|\phi| \log(|\phi|))}}$.

$\mathsf{LTL_{MP}}$ *Realizability and Synthesis –* Consider a finite set P partitioned as $I \uplus O$. Let $\mathsf{Lit}(P)$ be the set $\{p \mid p \in P\} \cup \{\neg p \mid p \in P\}$ of literals over P, and let $w : \mathsf{Lit}(P) \to \mathbb{Z}$ be a *weight function* where positive numbers represent rewards[1]. For all $S \in \{I, O\}$, this function is extended to Σ_S by: $w(\sigma) = \Sigma_{p \in \sigma} w(p) + \Sigma_{p \in S \setminus \{\sigma\}} w(\neg p)$ for $\sigma \in \Sigma_S$. It can also be extended to Σ_P as $w(o \cup i) = w(o) + w(i)$ for all $o \in \Sigma_O$ and $i \in \Sigma_I$. In the sequel, we denote by $\langle P, w \rangle$ the pair given by the finite set P and the weight function w over $\mathsf{Lit}(P)$; we also use the weighted alphabet $\langle \Sigma_P, w \rangle$.

Consider an LTL formula ϕ over $\langle P, w \rangle$ and an outcome $u = (o_0 \cup i_0)(o_1 \cup i_1) \cdots \in \Sigma_P^\omega$ produced by Players I and O. We associate a *value* $\mathsf{Val}(u)$ with u that captures the two objectives of Player O of both satisfying ϕ and achieving a mean-payoff objective. For each $n \geq 0$, let $u(n)$ be the prefix of u of length n. We define the *energy level* of $u(n)$ as $\mathsf{EL}(u(n)) = \sum_{k=0}^{n-1} w(o_k) + w(i_k)$. We then assign to u a *mean-payoff value* equal to $\mathsf{MP}(u) = \liminf_{n \to \infty} \frac{1}{n} \mathsf{EL}(u(n))$. Finally we define the value of u as $\mathsf{Val}(u) = \mathsf{MP}(u)$ if $u \models \phi$, and $\mathsf{Val}(u) = -\infty$ otherwise.

Given an LTL formula ϕ over $\langle P, w \rangle$ and a threshold $\nu \in \mathbb{Q}$, the $\mathsf{LTL_{MP}}$ *realizability problem (resp.* $\mathsf{LTL_{MP}}$ *realizability problem under finite memory)* asks to decide whether there exists a strategy (resp. finite-memory strategy) λ_O of Player O such that for all strategies $\lambda_I \in \Pi_I$, $\mathsf{Val}(\mathsf{Outcome}(\lambda_O, \lambda_I)) \geq \nu$, in which case we say that ϕ is MP-*realizable (resp.* MP*-realizable under finite memory)*. The $\mathsf{LTL_{MP}}$ *synthesis problem* is to produce such a winning strategy λ_O. So the aim is to achieve two objectives: *(i)* realizing ϕ, *(ii)* having a long-run average reward greater than the given threshold.

Optimality – Given ϕ an LTL formula over $\langle P, w \rangle$, the *optimal value* (for Player O) is defined as $\nu_\phi = \sup_{\lambda_O \in \Pi_O} \inf_{\lambda_I \in \Pi_I} \mathsf{Val}(\mathsf{Outcome}(\lambda_O, \lambda_I))$. For a real-valued $\epsilon \geq 0$, a strategy λ_O of Player O is ϵ*-optimal* if $\mathsf{Val}(\mathsf{Outcome}(\lambda_O, \lambda_I)) \geq \nu_\phi - \epsilon$ against all

[1] We use weights at several places of this paper. In some statements and proofs, we take the freedom to use *rational* weights as it is equivalent up to rescaling. However we always assume that weights are integers encoded in binary for complexity results.

strategies λ_I of Player I. It is *optimal* if it is ϵ-optimal with $\epsilon = 0$. Notice that ν_ϕ is equal to $-\infty$ if Player O cannot realize ϕ.

Example 2. Let us come back to Example 1 of a client-server system with two clients sharing a resource. The specification have been formalized by an LTL formula ϕ over the alphabet $P{=}I \uplus O$, with $I{=}\{r_1, r_2\}$, $O{=}\{g_1, w_1, g_2, w_2\}$. Suppose that we want to add the following constraints: client 2's requests take the priority over client 1's requests, but client 1's should still be eventually granted. Moreover, we would like to keep minimal the delay between requests and grants. This latter requirement has more the flavor of an optimality criterion and is best modeled using a weight function and a mean-payoff objective. To this end, we impose penalties to the waiting signals w_1, w_2, with a larger penalty to w_2 than to w_1. We thus use the following weight function $w : \mathsf{Lit}(P) \to \mathbb{Z}$: $w(w_1) = -1, w(w_2) = -2$ and $w(l) = 0, \forall l \notin \{w_1, w_2\}$.

One optimal strategy for the server is as follows: it almost always grants the resource to client 2 immediately after r_2 is set to true by client 2, and with a decreasing frequency grants request r_1 emitted by client 1. Such a server ensures a mean-payoff value equal to -1 against the most demanding behavior of the clients (where they are constantly requesting the shared resource). Such a strategy requires the server to use an infinite memory as it has to grant client 1 with an infinitely decreasing frequency. Note that a server that would grant client 1 in such a way without the presence of requests by client 1 would still be optimal. No finite memory server can be optimal. Indeed, if the server is allowed to count only up to a fixed positive integer $k \in \mathbb{N}$, then the best that it can do is : grant immediatly any request by client 2 if the last ungranted request of client 1 has been emitted less than k steps in the past, otherwise grant the request of client 1. The mean-payoff value of this solution, in the worst-case (when the two clients always emit their respective request) is equal to $-(1 + \frac{1}{k})$. So, even if finite memory cannot be optimal, we can devise a finite-memory strategy that is ϵ-optimal for any $\epsilon > 0$.

$\mathsf{LTL_E}$ *Realizability and Synthesis –* For the proofs of this paper, we need to consider realizability and synthesis with *energy* (instead of mean-payoff) objectives. With the same notations as before, the $\mathsf{LTL_E}$ *realizability problem* is to decide whether ϕ is E-*realizable*, that is, whether there exists $\lambda_O \in \Pi_O$ and $c_0 \in \mathbb{N}$ such that for all $\lambda_I \in \Pi_I$, *(i)* $u = \mathsf{Outcome}(\lambda_O, \lambda_I) \models \phi$, *(ii)* $\forall n \geq 0, c_0 + \mathsf{EL}(u(n)) \geq 0$. We thus ask if there exists an *initial credit* c_0 such that the energy level of each prefix $u(n)$ remains positive. When ϕ is E-realizable, the $\mathsf{LTL_E}$ *synthesis problem* is to produce such a winning strategy λ_O. Finally, we define the *minimum initial credit* as the least value of c_0 for which ϕ is E-realizable. A strategy λ_O is *optimal* if it is winning for the minimum initial credit.

3 Computational Complexity of the $\mathsf{LTL_{MP}}$ Realizability Problem

In this section, we solve the $\mathsf{LTL_{MP}}$ realizability problem, and we establish its complexity. Our solution relies on a reduction to a mean-payoff parity game. The same complexity result holds for the $\mathsf{LTL_E}$ realizability problem.

Theorem 2. *The* $\mathsf{LTL_{MP}}$ *realizability problem is 2ExpTime-Complete.*

Before proving this result, we recall useful notions on game graphs.

Game Graphs – A *game graph* $G = (S, s_0, E)$ consists of a finite set $S = S_1 \uplus S_2$ partitioned into S_1 the states of Player 1, and S_2 the states of Player 2, an initial state s_0, and a set $E \subseteq S \times S$ of edges such that for all $s \in S$, there exists a state $s' \in S$ such that $(s, s') \in E$. A game on G starts from s_0 and is played in rounds as follows. If the game is in a state of S_1, then Player 1 chooses the successor state among the set of outgoing edges; otherwise Player 2 chooses the successor state. Such a game results in an infinite path $\rho = s_0 s_1 \dots s_n \dots$ (called a *play*), whose prefix $s_0 s_1 \dots s_n$ is denoted by $\rho(n)$. We denote by $\mathsf{Plays}(G)$ the set of all plays in G and by $\mathsf{Pref}(G)$ the set of all prefixes of plays in G. A *turn-based* game is a game graph G such that $E \subseteq (S_1 \times S_2) \cup (S_2 \times S_1)$, meaning that each game is played in rounds alternatively by Player 1 and Player 2.

Objectives – An *objective* for G is a set $\Omega \subseteq S^\omega$. Let $p : S \to \mathbb{N}$ be a *priority function* and $w : E \to \mathbb{Z}$ be a *weight function* where positive weights represent rewards. The *energy level* of a prefix $\gamma = s_0 s_1 \dots s_n$ of a play is $\mathsf{EL}_G(\gamma) = \sum_{i=0}^{n-1} w(s_i, s_{i+1})$, and the *mean-payoff value* of a play $\rho = s_0 s_1 \dots s_n \dots$ is $\mathsf{MP}_G(\rho) = \liminf_{n\to\infty} \frac{1}{n} \cdot \mathsf{EL}_G(\rho(n))$.[2] Given a play ρ, we denote by $\mathsf{Inf}(\rho)$ the set of states $s \in S$ that appear infinitely often in ρ. The following objectives Ω are considered in the sequel:

– *Safety objective*. Given a set $\alpha \subseteq S$, the safety objective is defined as $\mathsf{Safety}_G(\alpha) = \mathsf{Plays}(G) \cap \alpha^\omega$.
– *Parity objective*. The parity objective is defined as $\mathsf{Parity}_G(p) = \{\rho \in \mathsf{Plays}(G) \mid \min\{p(s) \mid s \in \mathsf{Inf}(\rho)\}$ is even$\}$.
– *Energy objective*. Given an initial credit $c_0 \in \mathbb{N}$, the energy objective is defined as $\mathsf{PosEnergy}_G(c_0) = \{\rho \in \mathsf{Plays}(G) \mid \forall n \geq 0 : c_0 + \mathsf{EL}_G(\rho(n)) \geq 0\}$.
– *Mean-payoff objective*. Given a threshold $\nu \in \mathbb{Q}$, the mean-payoff objective is defined as $\mathsf{MeanPayoff}_G(\nu) = \{\rho \in \mathsf{Plays}(G) \mid \mathsf{MP}_G(\rho) \geq \nu\}$.
– *Combined objective*. The *energy safety* objective $\mathsf{PosEnergy}_G(c_0) \cap \mathsf{Safety}_G(\alpha)$ (resp. *energy parity* objective $\mathsf{PosEnergy}_G(c_0) \cap \mathsf{Parity}_G(p)$, *mean-payoff parity* objective $\mathsf{MeanPayoff}_G(\nu) \cap \mathsf{Parity}_G(p)$) combines the requirements of energy and safety (resp. energy and parity, energy and mean-payoff) objectives.

When an objective Ω is imposed on G, we say that G is an Ω *game*. For instance, if Ω is an energy parity objective, we say that $\langle G, w, p \rangle$ is an *energy parity game*, aso.

Strategies – Given a game graph G, a *strategy* for Player 1 is a function $\lambda_1 : S^* S_1 \to S$ such that $(s, \lambda_1(\gamma \cdot s)) \in E$ for all $\gamma \in S^*$ and $s \in S_1$. A play $\rho = s_0 s_1 \dots s_n \dots$ starting from the initial state s_0 is compatible with λ_1 if for all $k \geq 0$ such that $s_k \in S_1$ we have $s_{k+1} = \lambda_1(\rho(k))$. Strategies and play compatibility are defined symmetrically for Player 2. The set of strategies of Player i is denoted by Π_i, $i{=}1, 2$. We denote by $\mathsf{Outcome}_G(\lambda_1, \lambda_2)$ the play from s_0, compatible with λ_1 and λ_2. We let $\mathsf{Outcome}_G(\lambda_1) = \{\mathsf{Outcome}_G(\lambda_1, \lambda_2) \mid \lambda_2{\in}\Pi_2\}$. A strategy $\lambda_1{\in}\Pi_1$ is *winning* for an objective Ω if $\mathsf{Outcome}_G(\lambda_1) \subseteq \Omega$. We also say that λ_1 is winning in the Ω game G.

A strategy λ_1 of Player 1 is *finite-memory* if there exists a right-congruence $\sim$ on $\mathsf{Pref}(G)$ with finite index such that $\lambda_1(\gamma \cdot s_1) = \lambda_1(\gamma' \cdot s_1)$ for all $\gamma \sim \gamma'$ and $s_1 \in S_1$. The *size* of the memory is the index of $\sim$.

[2] Notation EL, MP and $\mathsf{Outcome}$ is here used with the index G to avoid any confusion with the same notation introduced in the previous section.

Optimal Value and ϵ-Optimality – Let us turn to mean-payoff parity games $\langle G, w, p\rangle$. With each play $\rho \in \mathsf{Plays}(G)$, we associate a *value* $\mathsf{Val}_G(\rho)$ defined as follows:

$$\mathsf{Val}_G(\rho) = \begin{cases} \mathsf{MP}_G(\rho) & \text{if } \rho \in \mathsf{Parity}_G(p) \\ -\infty & \text{otherwise.} \end{cases}$$

We define $\nu_G = \sup_{\lambda_1 \in \Pi_1} \inf_{\lambda_2 \in \Pi_2} \mathsf{Val}_G(\mathsf{Outcome}_G(\lambda_1, \lambda_2))$ as the *optimal value* for Player 1. For a real-valued $\epsilon \geq 0$, a strategy $\lambda_1 \in \Pi_1$ is *ϵ-optimal* if we have $\mathsf{Val}_G(\mathsf{Outcome}_G(\lambda_1, \lambda_2)) \geq \nu_G - \epsilon$ for all strategies $\lambda_2 \in \Pi_2$. It is *optimal* if it is ϵ-optimal with $\epsilon = 0$. If Player 1 cannot achieve the parity objective, then $\nu_G = -\infty$, otherwise optimal strategies exist [11] and ν_G is the largest threshold ν for which Player 1 can hope to achieve $\mathsf{MeanPayoff}_G(\nu)$.

Theorem 3 ([11,6,12]). *The optimal value of a mean-payoff parity game $\langle G, w, p\rangle$ can be computed in time $O(|E| \cdot |S|^{d+2} \cdot W)$, where $|E|$ (resp. $|S|$) is the number of edges (resp. states) of G, d is the number of priorities of p, and W is the largest absolute weight used by w. When $\nu_G \neq -\infty$, optimal strategies for Player 1 may require infinite memory; however for all $\epsilon > 0$, Player 1 has a finite-memory ϵ-optimal strategy.*

Proof (of Theorem 2). The classical realizability procedure for plain LTL first transforms the LTL formula into a non-deterministic Büchi automaton and then into a deterministic parity automaton. This deterministic automaton directly defines a parity game in which Player 1 has a strategy iff Player O has a strategy to realize the LTL specification. For a LTL_MP specification ϕ, we follow the same path but extend the parity game into a mean-payoff parity game using the weight function w. It can be shown that the game we obtain has the following size: $2^{2^{O(|\phi| \log |\phi|)}}$ states and $2^{O(|\phi|)}$ priorities. By Theorem 3, we get the 2ExpTime upper bound for LTL_MP realizability. The lower bound is a direct consequence of 2ExpTime-hardness of (qualitative) LTL realizability. $\square$

Based on Theorem 3 and the proof of Theorem 2 we get the following results on ϵ-optimality and finite-memory strategies:

Corollary 4. *Let ϕ be an LTL formula. If ϕ is MP-realizable, then for all $\epsilon > 0$, Player O has an ϵ-optimal winning strategy that is finite-memory, that is*

$$\nu_\phi = \sup_{\substack{\lambda_O \in \Pi_O \\ \lambda_O \text{ finite-memory}}} \inf_{\lambda_I \in \Pi_I} \mathsf{Val}(\mathsf{Outcome}(\lambda_O, \lambda_I)).$$

Motivated by this result, we focus on finite-memory strategies in the sequel.

Solution to the LTL_E Realizability Problem – We solve the LTL_E realizability problem with a reduction to energy parity games for which the following theorem holds:

Theorem 5 ([8]). *Whether there exist an initial credit c_0 and a winning strategy for Player 1 in a given energy parity game $\langle G, w, p\rangle$ for c_0 can be decided in time $O(|E| \cdot d \cdot |S|^{d+3} \cdot W)$. Moreover if Player 1 wins, then he has a finite-memory winning strategy with a memory size bounded by $4 \cdot |S| \cdot d \cdot W$.*

As for $\mathsf{LTL}_{\mathsf{MP}}$, one can reduce $\mathsf{LTL}_{\mathsf{E}}$ realizability to energy parity games and show that $\mathsf{LTL}_{\mathsf{E}}$ realizability is 2ExpTime-Complete based on Theorem 5.

Theorem 6. *The $\mathsf{LTL}_{\mathsf{E}}$ realizability problem is 2ExpTime-Complete. Moreover, if a formula ϕ over $\langle P, w\rangle$ is E-realizable, then it is E-realizable by a finite-memory strategy with a memory size at most doubly-exponential in the size of the input, i.e. the LTL formula and the function w (with weights encoded in binary).*

The constructions proposed in Theorems 2 and 6 can be easily extended to the more general case where the weights assigned to executions are given by a deterministic weighted automaton, as proposed in [9], instead of a weight function w over $\mathsf{Lit}(P)$ as done here. Indeed, given an LTL formula ϕ and a deterministic weighted automaton $\mathcal{A}$, we first construct from ϕ a deterministic parity automaton and then take the synchronized product with $\mathcal{A}$. Finally this product can be turned into a mean-payoff (resp. energy) parity game.

4 Safraless Algorithm

In the previous section, we have proposed an algorithm for solving the $\mathsf{LTL}_{\mathsf{MP}}$ realizability of a given LTL formula ϕ, which is based on a reduction to a mean-payoff parity game denoted by G_ϕ. This algorithm has two main drawbacks. First, it requires the use of Safra's construction to get a deterministic parity automaton $\mathcal{A}_\phi$ such that $L(\mathcal{A}_\phi) = [\![\phi]\!]$, a construction which is resistant to efficient implementations [1]. Second, strategies for the game G_ϕ may require infinite memory (for the threshold ν_{G_ϕ}, see Theorem 3). This is also the case for the $\mathsf{LTL}_{\mathsf{MP}}$ realizability problem, as illustrated by Example 2. In this section, we show how to circumvent these two drawbacks.

The second drawback has been already partially solved by Corollary 4, when the threshold given for the $\mathsf{LTL}_{\mathsf{MP}}$-realizability is the optimal value ν_ϕ. Indeed it states the existence of finite-memory winning strategies for the thresholds $\nu_\phi - \epsilon$, for all $\epsilon > 0$. We here show that we can go further by translating the $\mathsf{LTL}_{\mathsf{MP}}$ realizability problem under finite memory into an $\mathsf{LTL}_{\mathsf{E}}$ realizability problem, and conversely, by shifting the weights by the threshold value [8]:

Theorem 7. *An LTL formula ϕ over a weighted alphabet $\langle P, w\rangle$ is MP-realizable under finite memory for a threshold $\nu \in \mathbb{Q}$ iff ϕ over the weighted alphabet $\langle P, w-\nu\rangle$ is E-realizable.*

It is important to notice that when we want to synthesize ϵ-optimal strategies for $\mathsf{LTL}_{\mathsf{MP}}$ by reduction to $\mathsf{LTL}_{\mathsf{E}}$, the memory size of the strategy increases as ϵ decreases. Indeed, if $\epsilon = \frac{a}{b}$, then the weight function (for $\mathsf{LTL}_{\mathsf{E}}$ realizability) must be multiplied by b in a way to have integer weights (see footnote 1). The largest absolute weight W is thus also multiplied by b.

To avoid the Safra's construction needed to obtain a deterministic parity automaton for the underlying LTL formula, we adapt a Safraless construction proposed in [20,14] for the LTL synthesis problem, in a way to deal with weights and efficiently solve the $\mathsf{LTL}_{\mathsf{E}}$ synthesis problem. Instead of constructing a mean-payoff parity game from a deterministic parity automaton, we propose a reduction to a safety game. In this aim, we need to define the notion of energy automaton.

Energy Automata – Let $\langle P, w\rangle$ with P a finite set of signals and w a weight function over $\mathsf{Lit}(P)$. We are going to recall several notions of automata on infinite words over Σ_P and introduce the related notion of energy automata over the weighted alphabet $\langle \Sigma_P, w\rangle$. An *automaton* $\mathcal{A}$ over the alphabet Σ_P is a tuple $(\Sigma_P, Q, q_0, \alpha, \delta)$ such that Q is a finite set of states, $q_0 \in Q$ is the initial state, $\alpha \subseteq Q$ is a set of final states and $\delta : Q \times \Sigma_P \to 2^Q$ is a transition function. We say that $\mathcal{A}$ is *deterministic* if $\forall q \in Q, \forall \sigma \in \Sigma_P, |\delta(q, \sigma)| \leq 1$. It is *complete* if $\forall q \in Q, \forall \sigma \in \Sigma_P, \delta(q, \sigma) \neq \varnothing$.

A *run* of $\mathcal{A}$ on a word $u = \sigma_0\sigma_1 \cdots \in \Sigma_P^\omega$ is an infinite sequence of states $\rho = \rho_0\rho_1 \cdots \in Q^\omega$ such that $\rho_0 = q_0$ and $\forall k \geq 0, \rho_{k+1} \in \delta(\rho_k, \sigma_k)$. We denote by $\mathsf{Runs}_{\mathcal{A}}(u)$ the set of runs of $\mathcal{A}$ on u, and by $\mathsf{Visit}(\rho, q)$ the number of times the state q occurs along the run ρ. We consider the following acceptance conditions:

Non-deterministic Büchi:	$\exists \rho \in \mathsf{Runs}_{\mathcal{A}}(u), \exists q \in \alpha, \mathsf{Visit}(\rho, q) = \infty$
Universal co-Büchi:	$\forall \rho \in \mathsf{Runs}_{\mathcal{A}}(u), \forall q \in \alpha, \mathsf{Visit}(\rho, q) < \infty$
Universal K-co-Büchi:	$\forall \rho \in \mathsf{Runs}_{\mathcal{A}}(u), \sum_{q\in\alpha} \mathsf{Visit}(\rho, q) \leq K.$

A word $u \in \Sigma_P^\omega$ is *accepted* by a *non-deterministic Büchi automaton* (NB) $\mathcal{A}$ if u satisfies the non-deterministic Büchi acceptance condition. We denote by $L_{\text{nb}}(\mathcal{A})$ the set of words accepted by $\mathcal{A}$. Similarly we have the notion of *universal co-Büchi automaton* (UCB) $\mathcal{A}$ (resp. *universal K-co-Büchi automaton* (U*K*CB) $\langle \mathcal{A}, K\rangle$) and the set $L_{\text{ucb}}(\mathcal{A})$ (resp. $L_{\text{ucb},K}(\mathcal{A})$) of accepted words.

We now introduce energy automata. Let $\mathcal{A}$ be a NB over the alphabet Σ_P. The related *energy non-deterministic Büchi automaton* (eNB) $\mathcal{A}^w$ is over the weighted alphabet $\langle \Sigma_P, w\rangle$ and has the same structure as $\mathcal{A}$. Given an initial credit $c_0 \in \mathbb{N}$, a word u is *accepted* by $\mathcal{A}^w$ if *(i)* u satisfies the non-deterministic Büchi acceptance condition and *(ii)* $\forall n \geq 0, c_0 + \mathsf{EL}(u(n)) \geq 0$. We denote by $L_{\text{nb}}(\mathcal{A}^w, c_0)$ the set of words accepted by $\mathcal{A}^w$ with the given initial credit c_0. We also have the notions of *energy universal co-Büchi automaton* (eUCB) $\mathcal{A}^w$ and *energy universal K-co-Büchi automaton* (eU*K*CB) $\langle \mathcal{A}^w, K\rangle$, and the related sets $L_{\text{ucb}}(\mathcal{A}^w, c_0)$ and $L_{\text{ucb},K}(\mathcal{A}^w, c_0)$. Notice that if $K \leq K'$ and $c_0 \leq c_0'$, then $L_{\text{ucb},K}(\mathcal{A}^w, c_0) \subseteq L_{\text{ucb},K'}(\mathcal{A}^w, c_0')$.

The interest of U*K*CB is that they can be determinized with the subset construction extended with counters [14,20]. This construction also holds for eU*K*CB by using counting functions F. Intuitively, for all states q of $\mathcal{A}^w$, with $F(q)$ we count (up to $K + 1$) the maximal number of accepting states which have been visited by runs ending in q. The counter $F(q)$ is equal to -1 when no run ends in q. The final states are counting functions F such that $F(q) > K$ for some state q (accepted runs avoid such F).

It results in a deterministic automaton that we denote $\det(\mathcal{A}^w, K)$ and which has the following properties:

Proposition 8. *Let $K \in \mathbb{N}$ and $\langle \mathcal{A}^w, K\rangle$ be an* eU*K*CB. *Then $\det(\mathcal{A}^w, K)$ is a deterministic and complete energy automaton such that $L_{ucb,0}(\det(\mathcal{A}^w, K), c_0) = L_{ucb,K}(\mathcal{A}^w, c_0)$ for all $c_0 \in \mathbb{N}$.*

Our Safraless solution relies on the following theorem:

Theorem 9. *Let ϕ be an* LTL *formula over $\langle P, w_P\rangle$. Let $\langle G_\phi, w, p\rangle$ be the associated energy parity game with $|S|$ being its the number of states, d its number of priorities and W its largest absolute weight. Let $\mathcal{A}$ be a* UCB *with n states such that $L_{ucb}(\mathcal{A}) = [\![\phi]\!]$.*

Let $K = 4 \cdot n \cdot |S|^2 \cdot d \cdot W$ and $C = K \cdot W$. Then ϕ is E*-realizable iff there exists a Moore machine $\mathcal{M}$ such that $L(\mathcal{M}) \subseteq L_{ucb,K}(\mathcal{A}^w, C)$.*

Proof. Theorem 6 tells us that ϕ is E-realizable iff there exists a Moore machine $\mathcal{M}$ such that $L(\mathcal{M}) \subseteq L_{\mathsf{ucb}}(\mathcal{A}^w, c_0)$ for some $c_0 \geq 0$ and $|\mathcal{M}| = 4 \cdot |S|^2 \cdot d \cdot W$. Consider now the product of $\mathcal{M}$ and $\mathcal{A}^w$: in any accessible cycle of this product, there is no accepting state of $\mathcal{A}^w$ (as shown similarly for the qualitative case [14]) and the sum of the weights must be positive. The length of a path reaching such a cycle is at most $n \cdot |\mathcal{M}|$, therefore one gets $L(\mathcal{M}) \subseteq L_{\mathsf{ucb},n\cdot|\mathcal{M}|}(\mathcal{A}^w, n \cdot |\mathcal{M}| \cdot W)$. $\square$

As we have seen before, the eU*K*CB $\mathcal{A}^w$ can be easily determinized and thus converted into an energy safety objective. By memorizing the energy levels up to C [7], this energy safety objective can be converted into a safety objective, and so we get:

Theorem 10. *Let ϕ be an* LTL *formula. Then one can construct a safety game in which Player 1 has a winning strategy iff ϕ is* E*-realizable.*

5 Extension to Multi-dimensional Weights

Multi-Dimensional $\mathsf{LTL_{MP}}$ *and* $\mathsf{LTL_E}$ *Realizability Problems –* The $\mathsf{LTL_{MP}}$ and $\mathsf{LTL_E}$ realizability problems can be naturally extended to multi-dimensional weights. Given P, we define a weight function $w : \mathsf{Lit}(P) \to \mathbb{Z}^m$, for some dimension $m \geq 1$. The concepts of energy level EL, mean-payoff value MP, and value Val are defined similarly. Given an LTL formula ϕ over $\langle P, w\rangle$ and a threshold $\nu \in \mathbb{Q}^m$, the *multi-dimensional* $\mathsf{LTL_{MP}}$ *realizability problem under finite memory* asks to decide whether there exists a Player O's finite-memory strategy λ_O such that $\mathsf{Val}(\mathsf{Outcome}(\lambda_O, \lambda_I)) \geq$[3] ν against all strategies $\lambda_I \in \Pi_I$. The *multi-dimensional* $\mathsf{LTL_E}$ *realizability problem* asks to decide whether there exists $\lambda_O \in \Pi_O$ and an initial credit $c_0 \in \mathbb{N}^m$ such that for all $\lambda_I \in \Pi_I$, *(i)* $u = \mathsf{Outcome}(\lambda_O, \lambda_I) \models \phi$, *(ii)* $\forall n \geq 0,\ c_0 + \mathsf{EL}(u(n)) \geq (0, \dots, 0)$.

Computational Complexity – The 2ExpTime-completeness of the $\mathsf{LTL_{MP}}$ and $\mathsf{LTL_E}$ realizability problems have been stated in Theorem 2 and 6 in one dimension. In the multi-dimensional case, we have the next result.

Theorem 11. *The multi-dimensional* $\mathsf{LTL_{MP}}$ *realizability problem under finite memory and the multi-dimensional* $\mathsf{LTL_E}$ *realizability problem are in co-N2ExpTime.*

Proof. As for the one-dimensional case, $\mathsf{LTL_{MP}}$ realizability problem under finite memory and the multi-dimensional $\mathsf{LTL_E}$ realizability problem are inter-reducible by substracting the threshold to the weight values. So let us focus on $\mathsf{LTL_E}$ realizability. From the LTL formula we follow the same path as the one-dimensional case by constructing an equivalent deterministic parity automaton, that can be seen as a parity game. We add multi-weights to this game and so the $\mathsf{LTL_E}$ realizability problem amounts to solve a multi-energy parity game. Such games have been studied in [12], where it is shown how to remove the parity condition by adding extra dimensions in the game. This leads

[3] With $a \geq b$, we mean $a_i \geq b_i$ for all i, $1 \leq i \leq m$.

to resolving a multi-energy game, which can be done with a co-NPTime procedure, as shown in [10]. As this procedure executes on a doubly exponential game, we get the co-N2ExpTime upper bound. □

The Safraless procedure that we propose in one dimension can be extended to this multi-dimensional setting using recent results obtained in [12].

6 Implementation and Experiments

In the previous sections, in one or several dimensions, we have shown how to reduce the $\mathsf{LTL}_{\mathsf{MP}}$ under finite memory and $\mathsf{LTL}_{\mathsf{E}}$ realizability problems to safety games. We first discuss how antichain techniques can be used to symbolically solve those safety games. This approach has been implemented in our tool Acacia+. We then briefly present this tool and give some experimental results.

Antichain-Based Algorithm – Safety games with the objective $\mathsf{Safety}_G(\alpha)$ can be solved backwardly by computing the fixpoint of the following sequence: $W_0{=}\alpha$ and for all $k \geq 0$, $W_{k+1}{=}W_k \cap \{\{s{\in}S_1 \mid \exists(s,s'){\in}E, s' \in W_k\} \cup \{s{\in}S_2 \mid \forall(s,s'){\in}E, s' \in W_k\}\}$.

Therefore one needs to manipulate sets of states. The states of the safety game in our Safraless procedure are tuples (F, c) where F is a counting function as described before Proposition 8 and c is an energy level. They can be ordered as follows: $(F_1, c_1) \preceq (F_2, c_2)$ iff $F_1(q) \leq F_2(q)$ for all automata state q and $c_1 \geq c_2$. Intuitively, if Player 1 can win from (F_2, c_2) then he can win from (F_1, c_1), as he has seen more accepting states and has less energy in (F_2, c_2) than in (F_1, c_1). The sets of the sequence $(W_k)_k$ are all closed for that partial order, and can thus be represented by the antichain of their maximal elements, following ideas of [14]. We also exploit this order in a forward algorithm for solving safety games as done in [14].

Incrementality Approach – The size of the parameters K and C ensuring completeness (see Theorem 9) are doubly exponential, and this is clearly impractical. Nevertheless, we can use the following property: $L_{\text{ucb},K_1}(\mathcal{A}^w, C_1) \subseteq L_{\text{ucb},K_2}(\mathcal{A}^w, C_2)$ for all $C_1 \leq C_2$ and $K_1 \leq K_2$. This inclusion tells us that if there exists a Moore machine $\mathcal{M}$ such that $L(\mathcal{M}) \subseteq L_{\text{ucb},K_1}(\mathcal{A}^w, C_1)$ then the formula is E-realizable without considering the huge theoretical bounds K and C of Theorem 9. This means that we can adopt as in [14], an incremental approach that first uses small values for parameters K and C and increments them when necessary (if the more constrained specification is not realizable).

Tool Acacia+ – In [3] we present Acacia+, a tool for LTL synthesis using antichain-based algorithms. Its main advantage, regarding other LTL synthesis tools, is to generate compact strategies that are usable in practice. This can be very useful in applications like control code synthesis from high-level LTL specifications, debugging of unrealizable LTL specifications by inspecting compact counter strategies, and generation of small deterministic Büchi or parity automata from LTL formulas (when they exist) [3].

Acacia+ is now extended to the synthesis from LTL specifications with mean-payoff objectives in the multi-dimensional setting. As explained before, it solves incrementally

Table 1. Acacia+ on the specification of Example 2 with increasing threshold values ν. The column K (resp. C) gives the minimum value (resp. vector) required to obtain a winning strategy, M the size of the finite-memory strategy, $time$ the execution time (in seconds) and mem the total memory usage (in megabytes). Note that the running time is the execution time of the forward algorithm applied to the safety game with values K and C (and not with smaller ones).

ν	-1.2	-1.02	-1.002	-1.001	-1.0002	-1.0001	-1.00005
K	4	49	499	999	4999	9999	19999
C	7	149	1499	2999	14999	29999	99999
M	5	50	500	1000	5000	10000	20000
time (s)	0.01	0.05	0.34	0.89	15.49	59.24	373
mem (MB)	9.75	9.88	11.29	12.58	30	48.89	86.68

a family of safety games, depending on some values K and C, to test whether a given specification ϕ is MP-realizable under finite memory. The tool takes as input an LTL formula ϕ with a partition of its set P of atomic signals, a weight function $w : \mathsf{Lit}(P) \mapsto \mathbb{Z}^m$, a threshold value $\nu \in \mathbb{Q}^m$, and two bounds $K \in \mathbb{Z}$ and $C \in \mathbb{Z}^m$ (the user can specify additional parameters to define the incremental policy). It then searches for a finite-memory winning strategy for Player O, within the bounds K and C, and outputs a Moore machine if such a strategy exists. The last version of Acacia+, a web interface for using it online, some benchmarks and experimental results can be found at `http://lit2.ulb.ac.be/acaciaplus/`.

Experiments – We now present some experiments. They have been done on a Linux platform with a 3.2GHz CPU (Intel Core i7) and 12GB of memory.

(1) Approaching the optimal value. Consider the specification ϕ of Example 2 and its 1-dimensional mean-payoff objective. We have shown that infinite memory strategies are required for the optimal value -1, but finite-memory ϵ-optimal strategies exist for all $\epsilon > 0$. In Table 1, we present the experiments done for some values of $-1-\epsilon$.

The strategies for the system output by Acacia+ are: grant the second client $(M-1)$ times, then grant once client 1, and start over. Thus, the system almost always plays $g_2 w_1$, except every M steps where he has to play $g_1 w_2$. Obviously, these strategies are the smallest ones that ensure the corresponding threshold values. They can also be compactly represented by a two-state automaton with a counter that counts up to M. Let us emphasize the interest of using antichains. With $\nu = -1.001$, the underlying state space manipulated by our symbolic algorithm has a huge size: around 10^{27}, since $K = 999$, $C = 2999$ and the number of automata states is 8. However the fixpoint computed backwardly is represented by an antichain of size 2004 only.

(2) No unsollicited grants. The major drawback of the strategies presented in Table 1 is that many unsollicited grants might be sent as the strategies do not take into account client requests, and just grant the resource access to the clients in a round-robin fashion (with a longer access for client 2). It is possible to express in LTL the absence of unsollicited grants, but it is cumbersome. Alternatively, the LTL_{MP} specification can be easily rewritten with a multi-dimensional mean-payoff objective. The specification of Examples 1 and 2 can be indeed extended with a new dimension per client, such that a request

Table 2. Acacia+ on the Shared Resource Arbiter benchmark parameterized by the number of clients, with the forward algorithm. The column c gives the number of clients, ν the threshold, K (resp. C) the minimum value (resp. vector) required to obtain a winning strategy, M the size of the finite-memory strategy, $time$ the total execution time (in seconds) and mem the total memory usage (in megabytes).

c	ν	K	C	$\lvert\mathcal{M}\rvert$	time (s)	mem (MB)
2	$(-1.2, 0, 0)$	4	$(7, 1, 1)$	11	0.02	10.04
3	$(-2.2, 0, 0, 0)$	9	$(19, 1, 1, 1)$	27	0.22	10.05
4	$(-3.2, 0, 0, 0, 0)$	14	$(12, 1, 1, 1, 1)$	65	1.52	12.18
5	$(-4.2, 0, 0, 0, 0, 0)$	19	$(29, 1, 1, 1, 1, 1)$	240	48	40.95
6	$(-5.2, 0, 0, 0, 0, 0, 0)$	24	$(17, 1, 1, 1, 1, 1, 1)$	1716	3600	636

(resp. grant) signal of client i has a reward (resp. cost) of 1 on his new dimension. More precisely, the weight function is now $w : \mathsf{Lit}(P) \to \mathbb{Z}^3$ such that $w(r_1) = (0, 1, 0)$, $w(r_2) = (0, 0, 1)$, $w(g_1) = (0, -1, 0)$, $w(g_2) = (0, 0, -2)$, $w(w_1) = (-1, 0, 0)$, $w(w_2) = (-2, 0, 0)$ and $w(l) = (0, 0, 0)$, $\forall l \in \mathsf{Lit}(P) \setminus \{r_1, r_2, g_1, g_2, w_1, w_2\}$. For $\nu = (-1, 0, 0)$, there is no hope to have a finite-memory strategy (see Example 2). For $\nu = (-1.2, 0, 0)$, Acacia+ outputs a finite-memory strategy of size 8 (with the backward algorithm) that prevents unsollicited grants. Moreover, this is the smallest strategy that ensures this threshold.

From the latter example we derive a benchmark of multi-dimensional examples parameterized by the number of clients making requests to the server. Some experimental results of Acacia+ on this benchmark are reported in Table 2.

(3) Approching the Pareto curve. As last experiment, we consider again the 2-client request-grant example with the weight function $w(w_1) = (-1, 0, 0, 0)$ and $w(w_2) = (0, -2, 0, 0)$. For this new specification there are several optimal values (w.r.t. the pairwise order), corresponding to trade-offs between the two objectives that are (i) to quickly grant client 1 and (ii) to quickly grant client 2. We try to approach, by hand, the *Pareto curve*, which consists of all those optimal values, i.e. to find finite-memory

Table 3. Acacia+ to approach Pareto values. The column ν gives the threshold, relatively close to the Pareto curve, K (resp. C) the minimum value (resp. vector) required to obtain a winning strategy, M the memory size of the strategy.

ν	K	C	M
$(-0.001, -2, 0, 0)$	999	$(1999, 1, 1, 1)$	2001
$(-0.15, -1.7, 0, 0)$	55	$(41, 55, 1, 1)$	42
$(-0.25, -1.5, 0, 0)$	3	$(7, 9, 1, 1)$	9
$(-0.5, -1, 0, 0)$	1	$(3, 3, 1, 1)$	5
$(-0.75, -0.5, 0, 0)$	3	$(9, 7, 1, 1)$	9
$(-0.85, -0.3, 0, 0)$	42	$(55, 41, 1, 1)$	9
$(-1, -0.01, 0, 0)$	199	$(1, 399, 1, 1)$	401

strategies that are incomparable w.r.t. the ensured thresholds, these thresholds being as large as possible. We give some such thresholds in Table 3, along with minimum K and C and strategy sizes. It is difficult to automatize the construction of the Pareto curve. Indeed, Acacia+ cannot test (in reasonable time) whether a formula is MP-unrealizable for a given threshold, since it has to reach the huge theoretical bound on K and C. This raises two interesting questions that we let as future work: how to decide efficiently that a formula is MP-unrealizable for a given threshold, and how to compute points of the Pareto curve efficiently.

References

1. Althoff, C.S., Thomas, W., Wallmeier, N.: Observations on determinization of Büchi automata. Theor. Comput. Sci. 363(2), 224–233 (2006)
2. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better Quality in Synthesis through Quantitative Objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
3. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.-F.: Acacia+, a Tool for LTL Synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 652–657. Springer, Heidelberg (2012)
4. Boker, U., Chatterjee, K., Henzinger, T.A., Kupferman, O.: Temporal specifications with accumulative values. In: LICS, pp. 43–52. IEEE Computer Society (2011)
5. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite Runs in Weighted Timed Automata with Energy Constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
6. Bouyer, P., Markey, N., Olschewski, J., Ummels, M.: Measuring Permissiveness in Parity Games: Mean-Payoff Parity Games Revisited. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 135–149. Springer, Heidelberg (2011)
7. Brim, L., Chaloupka, J., Doyen, L., Gentilini, R., Raskin, J.-F.: Faster algorithms for mean-payoff games. Formal Methods in System Design 38(2), 97–118 (2011)
8. Chatterjee, K., Doyen, L.: Energy Parity Games. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 599–610. Springer, Heidelberg (2010)
9. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. ACM Trans. Comput. Log. 11(4) (2010)
10. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: FSTTCS. LIPIcs, vol. 8, pp. 505–516. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
11. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean-payoff parity games. In: LICS, pp. 178–187. IEEE Computer Society (2005)
12. Chatterjee, K., Randour, M., Raskin, J.-F.: Strategy Synthesis for Multi-Dimensional Quantitative Objectives. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 115–131. Springer, Heidelberg (2012)
13. Ehlers, R.: Symbolic Bounded Synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 365–379. Springer, Heidelberg (2010)
14. Filiot, E., Jin, N., Raskin, J.-F.: Antichains and compositional algorithms for LTL synthesis. Formal Methods in System Design 39(3), 261–296 (2011)
15. Henzinger, T.A.: Quantitative Reactive Models. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 1–2. Springer, Heidelberg (2012)

16. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE Computer Society (2006)
17. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS, pp. 531–542. IEEE Computer Society (2005)
18. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer (1992)
19. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM Press (1989)
20. 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)
21. Tsai, M.-H., Fogarty, S., Vardi, M.Y., Tsay, Y.-K.: State of Büchi Complementation. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 261–271. Springer, Heidelberg (2011)
22. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theor. Comput. Sci. 158(1&2), 343–359 (1996)

PRISM-games: A Model Checker for Stochastic Multi-Player Games

Taolue Chen[1], Vojtěch Forejt[1], Marta Kwiatkowska[1], David Parker[2], and Aistis Simaitis[1]

[1] Department of Computer Science, University of Oxford, UK
[2] School of Computer Science, University of Birmingham, UK

Abstract. We present PRISM-games, a model checker for stochastic multi-player games, which supports modelling, automated verification and strategy synthesis for probabilistic systems with competitive or co-operative behaviour. Models are described in a probabilistic extension of the Reactive Modules language and properties are expressed using rPATL, which extends the well-known logic ATL with operators to reason about probabilities, various reward-based measures, quantitative properties and precise bounds. The tool is based on the probabilistic model checker PRISM, benefiting from its existing user interface and simulator, whilst adding novel model checking algorithms for stochastic games, as well as functionality to synthesise optimal player strategies, explore or export them, and verify other properties under the specified strategy.

1 Introduction

Stochastic games are a natural model for systems that exhibit probabilistic behaviour and which contain components that may either compete or cooperate in order to achieve a certain goal. The model has a rich underlying theory and applications in areas as diverse as economics and biology. Stochastic games also have many applications in computer science. Game-theoretic models of competitive or collaborative behaviour can be used to model, for example, distributed systems, security protocols or sensor networks; furthermore, many such systems are inherently probabilistic, e.g. due to failures or randomisation.

For simpler model subclasses, various verification tools are available and widely used. For probabilistic models such as Markov chains or Markov decision processes and their variants, probabilistic model checking tools like PRISM [9] and MRMC [8] provide verification of quantitative properties in probabilistic temporal logics. For (non-stochastic) games, model checkers such as MCMAS [10] and MOCHA [1] verify properties in ATL or epistemic logics. GAVS+ [7] is a general-purpose algorithmic game solver which includes support for simple stochastic games. Game-based verification tools also have applications to scheduling and synthesis problems, for example using timed games [2], qualitative stochastic games [3] or mean-payoff games [4].

In this paper, we present *PRISM-games*, which is, to the best of our knowledge, the first tool to provide modelling and quantitative verification for *stochastic multi-player games* (SMGs). The games are specified using an extension of the existing PRISM modelling language, which is a guarded-command based language inspired by the Reactive Modules formalism. Properties are specified in the temporal logic rPATL [5], which combines features of the multi-agent logic ATL, the probabilistic logic PCTL, as well as operators to reason about several different notions of reward/cost measures, numerical properties and precise probability values [6]. Currently, PRISM-games supports turn-based, perfect-information SMGs; future work will investigate efficient techniques for more general problem classes such as concurrent games and partial information. Turn-based games have, though, already proved to be sufficient to model, analyse and detect potential weaknesses systems in algorithms for energy management and collective decision making for autonomous systems [5].

PRISM-games builds upon the code-base of the existing PRISM model checker, extending existing features to provide a modelling language for stochastic multi-player games and a graphical user interface with model editor, discrete-event simulator and graph-plotting functionality. The core functionality of the new tool comprises novel methods for verifying quantitative properties of stochastic games [5,6]; and support for synthesising optimal player strategies, exploring or exporting them, and verifying other properties under the specified strategy.

2 Modelling Stochastic Multi-Player Games

A (turn-based) stochastic multi-player game (SMG) comprises a finite set of *players* and a finite set of *states*. In each state, exactly one player chooses (possibly randomly) from a set of available *probabilistic transitions* to determine the next state. To reason about SMGs, we use *strategies*, which determine the choices of transitions made by each player, based on the execution of the game so far.

In PRISM-games, SMGs are described in a modelling language similar to the Reactive Modules formalism. A model is composed of *modules*, whose state is determined by a set of *variables* and whose behaviour is specified by a set of *guarded commands*, containing an (optional) action label, a guard and a probabilistic update for the module's variables:

```
[action] guard -> prob₁ : update₁ + ... + probₙ : updateₙ;
```

When a module has a command whose guard is satisfied in the current state, it can update its variables probabilistically, accordingly to the update. For action-labelled commands, multiple modules execute updates synchronously, if all their guards are satisfied. Each probabilistic transition in the model is thus associated with either an action label or a single module. A model also defines *players*, each of which is assigned a disjoint subset of the model's synchronising action labels and modules. This assigns each probabilistic transition to one player. Currently, to ensure a turn-based SMG, all possible probabilistic transitions in a state must belong to the same player; the tool detects and disallows concurrent actions.

An excerpt from a PRISM-games model of *futures market investors* is shown in Fig. 1. There are 3 players: 2 investors and the market. At the start of each month, *investors* decide whether to invest or not; the *market* can decide to bar

```
smg

// Player definitions
player investor1
    [invest1], [noinvest1], [cashin1]
endplayer
player investor2
    [invest2], [noinvest2], [cashin2]
endplayer
player market
    [nobar], [bar1],  [bar2], sched, [month], [done]
endplayer

// Investor 1
module investor1
    // State: 0 = no reservation
    // 1 = made reservation
    // 2 = finished
    i1 : [0..2];
    // Decide whether invest or not
    [noinvest1] i1=0 | i1=1 & b1=1  →  (i1′=0);
    [invest1] i1=0 | i1=1 & b1=1  →  (i1′=1);
    // Cash in shares (if not barred)
    [cashin1] i1=1 & b1=0  →  (i1′=2);
    // Finished
    [done] i1=2 | term=1 → true;
endmodule

// Investor 2
module investor2 = investor1 [ . . . ] endmodule
```

```
// Market
module market
    // State: 0 = !barred, 1 = barred
    b1  : [0..1] init 1;
    b2  : [0..1] init 1;
    // Share value
    v  : [0..vmax] init vinit;
    // Bar one or none of the investors
    [nobar] true  →  (b1′=0) & (b2′=0);
    [bar1] b1=0  →  (b1′=1) & (b2′=0);
    [bar2] b2=0  →  (b2′=1) & (b1′=0);
    // Share price movement
    [month] true  →  p/10  :  (v′=up)
                + (1 − p/10)  :  (v′=down);
endmodule

// Scheduling module
module sched
    // Turn-based scheduling of players
endmodule

// Reward: Shares collection value
// for investor1, and both investors
rewards "profit1"
    [cashin1] i1=1  :  v;
endrewards
rewards "profit12"
    [cashin1] i1=1  :  v;
    [cashin2] i2=1  :  v;
endrewards
```

Fig. 1. Excerpt from a three-player game modelling *futures market investors*

one of the investors from investing and also picks the order in which the investors take decisions. Share price fluctuations are modelled as a random process. The full model, along with several larger examples, is available from [11].

3 Property Specification

PRISM-games' property specification language is based on rPATL [5]. rPATL is a CTL-style branching-time temporal logic used to express properties of SMGs, which combines the coalition operator $\langle\langle C\rangle\rangle$ of ATL, the probabilistic operator $\mathsf{P}_{\bowtie q}$ from PCTL, and an operator $\mathsf{R}^r_{\bowtie x}$ for reasoning about several types of expected reward/cost measures. The syntax of rPATL is given by the grammar:

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi\wedge\phi \mid \langle\langle C\rangle\rangle \mathsf{P}_{\bowtie q}[\psi] \mid \langle\langle C\rangle\rangle \mathsf{R}^r_{\bowtie x}[\mathsf{F}^\star\phi]$$
$$\psi ::= \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi \mid \phi\,\mathsf{U}^{\leq k}\,\phi \mid \mathsf{F}\,\phi \mid \mathsf{F}^{\leq k}\phi \mid \mathsf{G}\,\phi \mid \mathsf{G}^{\leq k}\phi$$

where a is an atomic proposition used to label SMG states, C is a *coalition* (a set of players), $\bowtie\in\{<,\leq,\geq,>\}$, $q\in\mathbb{Q}\cap[0,1]$, $x\in\mathbb{Q}_{\geq 0}$, r is a *reward structure* mapping states to non-negative rationals, $\star\in\{0,\infty,c\}$ and $k\in\mathbb{N}$.

An example rPATL formula is $\langle\langle\{1,2\}\rangle\rangle\mathsf{P}_{\geq 0.75}[\mathsf{F}^{\leq 5} goal]$, which means "players 1 and 2 have a (combined) strategy to ensure that the probability of reaching a '*goal*' state within 5 steps is at least 0.75, regardless of the strategies of other players in the game". The $\langle\langle C\rangle\rangle\mathsf{R}^r_{\bowtie x}[\mathsf{F}^\star\phi]$ operator is used in a similar fashion, but is annotated with a reward structure r and a type $\star \in \{0, \infty, c\}$. It states that coalition C has a strategy to ensure that expected amount of reward r cumulated until a ϕ-state is reached satisfies $\bowtie x$. The type $\star$ allows us to treat the case where ϕ is *not* reached differently, assigning zero reward ($\star$=0), infinite reward ($\star$=∞) or allowing reward to accumulate indefinitely ($\star$=c).

We support several extensions of rPATL, including 'quantitative' (numerical) operators, e.g., $\langle\langle C\rangle\rangle\mathsf{P}_{\max=?}[\psi]$, which gives the maximum probability of ψ that coalition C can guarantee, instead of a true/false value. PRISM-games also supports *precise* value operators $\langle\langle C\rangle\rangle\mathsf{P}_{=q}[\psi]$ and $\langle\langle C\rangle\rangle\mathsf{R}^{r}_{=x}[\mathsf{F}^{c}\phi]$ for *stopping* games (i.e., stochastic games where terminal states are reached with probability 1 under any pair of strategies), using the model checking algorithms of [6].

Examples. Some sample properties for the *futures market investor* model from the previous section (see Fig. 1) are provided below.

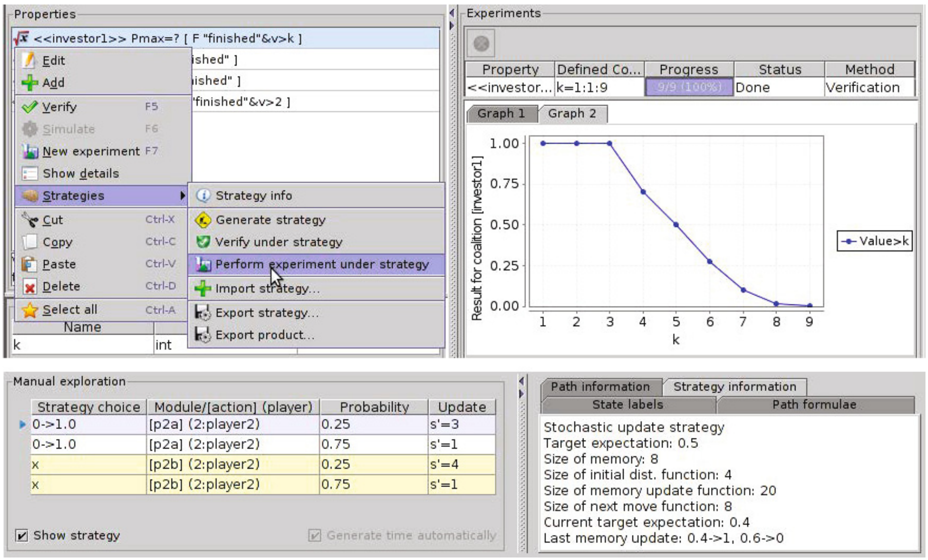
- $\langle\langle\{\text{investor1,investor2}\}\rangle\rangle\mathsf{R}^{\text{profit12}}_{\geq 10}[\mathsf{F}^{c}(\mathit{done1} \wedge \mathit{done2})]$ – "the two investors have a joint strategy guaranteeing them an expected profit of at least 10";
- $\langle\langle\{\text{investor1,market}\}\rangle\rangle\mathsf{R}^{\mathit{profit1}}_{\max=?}[\mathsf{F}^{c}\,\mathit{done1}]$ – "what is the maximum expected profit that investor 1 can achieve with the help of the market?";
- $\langle\langle\{\text{investor1,investor2}\}\rangle\rangle\mathsf{R}^{\text{profit12}}_{=5}[\mathsf{F}^{c}\,\mathit{done}]$ – "both investors can collaborate to achieve an expected profit of precisely 5";
- $\langle\langle\{\text{investor1}\}\rangle\rangle\mathsf{P}_{\max=?}[\mathsf{F}\,(\mathit{done1} \wedge v > 5)]$ – "what is the maximum probability with which investor 1 can guarantee a share value greater than 5?"

4 Synthesis and Analysis of Strategies

Reasoning about *strategies* is an essential aspect of modelling and verifying games. rPATL queries check for the existence of a strategy that satisfies a given probability/reward bound or which optimises some objective. When PRISM-games model checks such properties, it also supports *strategy synthesis*, allowing the user to obtain a corresponding satisfying/optimal strategy.

An SMG strategy resolves the choices in each state, using a (possibly infinite) set of *memory elements*, each representing a possible "state" of the strategy. The memory element is updated (possibly stochastically) at each transition, and the action picked by the player is determined by the current memory element and the current state. A strategy is *memoryless* if it has only one memory element, and *finite-memory* if there are finitely many. It is *deterministic* if the functions that update memory elements and pick actions are not probabilistic, and *stochastic-update* otherwise. Currently, PRISM-games supports three types of strategies: *memoryless deterministic*, *finite-memory deterministic* and *finite-memory stochastic-update*.

Strategies can be analysed manually in the simulator view or exported to files (see Fig. 2 for screenshots). PRISM-games also supports 'implementation' of strategies – the product of a strategy and the original game can be built, resulting in a new model on which other properties can be verified. For example, in a game with 3 players we can generate a strategy for player 1 specified



<<investor1>> Pmax=? [F "finished"&v>k]
ished"]
ished"]
'finished"&v>2]

Name	
k	int

Property	Defined Co...	Progress	Status	Method
<<investor...	k=1:1:9	9/9 (100%)	Done	Verification

Strategy choice	Module/[action] (player)	Probability	Update
0->1.0	[p2a] (2:player2)	0.25	s'=3
0->1.0	[p2a] (2:player2)	0.75	s'=1
x	[p2b] (2:player2)	0.25	s'=4
x	[p2b] (2:player2)	0.75	s'=1

Stochastic update strategy
Target expectation: 0.5
Size of memory: 8
Size of initial dist. function: 4
Size of memory update function: 20
Size of next move function: 8
Current target expectation: 0.4
Last memory update: 0.4->1, 0.6->0

Fig. 2. PRISM-games screenshots: simulation of a synthesised strategy (bottom) and verification of a property under the strategy (top)

by $\langle\langle 1 \rangle\rangle \mathsf{P}_{\geq 0.5}[\mathsf{F}\, goal1]$. Implementing this strategy would then result in a two-player game on which further properties may be verified, e.g., in rPATL formula $\langle\langle 2,3 \rangle\rangle \mathsf{P}_{\geq 0.99}[\mathsf{F}\, goal2]$, player 1 now does not minimise the probability of reaching a *goal2* state; instead its strategy is fixed to one which achieves the first rPATL formula. Strategy import functionality is also supported.

5 Algorithms and Further Details

Underlying Algorithms.. The core parts of the model checking algorithm for rPATL are based on reductions to two-player stochastic games, by constructing a *coalition game* where player 1 plays represents the coalition C from the rPATL formula being verified and player 2 is all other players. The basic techniques for solving games formulate systems of equations over $+, \cdot, \max, \min$, and then perform *value iteration* to compute their least or greatest solutions. This works directly for $\langle\langle C \rangle\rangle \mathsf{P}_{\bowtie q}[\psi]$ and $\langle\langle C \rangle\rangle \mathsf{R}^r_{\bowtie x}[\mathsf{F}^\star \phi]$ where $\star \in \{c, \infty\}$. For $\star = 0$, the optimal strategy may depend on the reward accumulated so far and so is not memoryless. Here, we compute a bound after which the optimal strategy picks actions that maximise the probability of reaching ϕ-states, and reduce the problem to previous cases; see [5] for details. For the *precise* value operators $\langle\langle C \rangle\rangle \mathsf{P}_{=q}[\psi]$ and $\langle\langle C \rangle\rangle \mathsf{R}^r_{=x}[\mathsf{F}^c \phi]$, we need to compute the sup inf and inf sup values for the property (using rPATL model checking algorithms). We then construct a finite-memory stochastic-update strategy based on the results of [6].

Table 1. Performance statistics for a representative set of models [5,11]

Case study		SMG statistics			Model checking		
		Players	States	Transitions	Property type	Constr.	Check
mdsm $[N]$	5	5	743,904	2,145,120	$\langle\langle C\rangle\rangle \mathsf{R}^{r}_{\max=?}[\mathsf{F}^{0}\,\phi]$	14.5s	61.9s
	7	7	6,241,312	19,678,246		210.7s	1,054.8s
cdmsn $[N]$	3	3	1,240	1,240	$\langle\langle C\rangle\rangle \mathsf{P}_{\bowtie q}[\,\mathsf{F}^{\leq k}\,\phi]$	0.2s	0.2s
	5	5	100,032	843,775		3.2s	6.4s
investor $[vmax]$	10	2	10,868	34,264	$\langle\langle C\rangle\rangle \mathsf{R}^{r}_{\min=?}[\mathsf{F}^{c}\,\phi]$	1.4s	0.7s
	200	2	2,931,643	9,688,354		45.9s	820.8s
team-form $[N]$	3	3	17,041	20,904	$\langle\langle C\rangle\rangle \mathsf{P}_{\max=?}[\mathsf{F}\,\phi]$	0.3s	0.5s
	5	5	2,366,305	2,893,536		36.9s	12.9s

Implementation and Availability. The model checking implementation is currently built upon PRISM's "explicit" engine, which uses Java-based explicit-state data structures (sparse matrices, bit-sets, etc.). Illustrative experimental results are shown in Table 1, for games of the order 10^6-10^7 states (run on a 2.8GHz PC with 32GB of RAM). A symbolic (BDD-based) implementation is under development to offer improved scalability on models exhibiting regularity.

PRISM-games is released as open source software, currently licensed under the GPL. The tool, with documentation and examples, is available from [11].

Acknowledgments. The authors are part supported by ERC Advanced Grant VERIWARE, the Institute for the Future of Computing at the Oxford Martin School, EPSRC grant EP/F001096/1 and a Royal Society Newton Fellowship.

References

1. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: MOCHA: Modularity in Model Checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for Playing Games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
3. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Radhakrishna, A.: Gist: A Solver for Probabilistic Games. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 665–669. Springer, Heidelberg (2010)
4. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: QUASY: Quantitative Synthesis Tool. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 267–271. Springer, Heidelberg (2011)
5. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: Automatic Verification of Competitive Stochastic Systems. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 315–330. Springer, Heidelberg (2012)
6. Chen, T., Forejt, V., Kwiatkowska, M., Simaitis, A., Trivedi, A., Ummels, M.: Playing Stochastic Games Precisely. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 348–363. Springer, Heidelberg (2012)
7. 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)

8. Katoen, J.P., Hahn, E.M., Hermanns, H., Jansen, D., Zapreev, I.: The ins and outs of the probabilistic model checker MRMC. In: Proc. QEST 2009. IEEE (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. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009)
11. PRISM-games website, `http://www.prismmodelchecker.org/games/`

PIC2LNT: Model Transformation for Model Checking an Applied Pi-Calculus

Radu Mateescu[1] and Gwen Salaün[2,1]

[1] Inria Grenoble Rhône-Alpes and LIG — CONVECS Team, France
[2] Grenoble INP, France
{Radu.Mateescu,Gwen.Salaun}@inria.fr

1 Introduction

The π-calculus [12] was proposed by Milner, Parrow, and Walker about twenty years ago for describing concurrent systems with mobile communication. The π-calculus is equipped with operational semantics defined in terms of LTSs (Labelled Transition Systems). Although a lot of theoretical results have been achieved on this language (see, *e.g.*, [1, chapter 8] for a survey), only a few verification tools have been designed for analysing π-calculus specifications automatically. The two most famous examples are the Mobility Workbench (MWB) [14] and JACK [5], which were developed in the 90s.

Our objective is to provide analysis features for π-calculus specifications by reusing the verification technology already available for value-passing process algebras without mobility. Contrary to existing verification tools for the π-calculus, which rely on specific algorithms and intermediate models, such as HD-automata [5], our approach is based on a novel translation [9] from the finite control fragment of π-calculus to a standard process algebra called LOTOS NT (LNT for short) [3]. LNT is a value-passing process algebra with imperative programming flavour accepted as input by the CADP verification toolbox [8]. It supports the specification of data structures (constructed types, pattern-matching, recursive functions) and concurrent processes. LNT has a user-friendly syntax and a formal operational semantics defined in terms of LTSs. To the best of our knowledge, this is the first π-calculus translation having a standard process algebra as target language.

In this work, we go a step further by extending the original polyadic π-calculus with data-handling features. This results in a general-purpose applied π-calculus, which offers a good level of expressiveness for specifying mobile concurrent systems, and therefore for widening its possible application domains. As language for describing data types and functions, a natural choice was LNT itself: in this way, the data types and functions used in the π-calculus specification can be directly imported into the LNT code produced by translation. We generalized our previous translation [9] to handle applied π-calculus specifications, and we automated it in the tool PIC2LNT 2.0. This enables the analysis of applied π-calculus specifications using all verification tools of CADP, in particular the EVALUATOR 4.0 on-the-fly model checker [11], which evaluates temporal properties involving channel names and data values.

2 Applied Pi-Calculus

We designed our applied π-calculus by extending the original polyadic π-calculus [12] equipped with the early operational semantics defined in [13]. We consider π-calculus specifications satisfying the *finite control* property [4], which amounts to forbid recursive agent calls through parallel composition operators. When the set of channels is bounded, this results in finite-state systems that can be analyzed using existing model checking techniques. We extended the original π-calculus with constructs for manipulating data variables and expressions. Agents can be parameterized by data variables in addition to channel names, and the polyadic communication was extended to handle emission/reception of data values. The guard operator was generalized to handle arbitrary Boolean expressions (in addition to the comparison of channel names), and a new operator was added for declaring and initializing data variables. The replication operator was restricted to a bounded version (in order to satisfy the finite control property), which instantiates n parallel copies of an agent, and therefore enables to describe mobile systems containing a finite amount of dynamic control. Data types and functions are specified in LNT [3] as external modules, which are imported in the applied π-calculus specification. The concrete syntax (which is compatible with MWB for dataless π-calculus specifications) and semantics of the applied π-calculus are described in [10].

We present below a code sample to illustrate our applied π-calculus on a load balancing system, which is a networking method to distribute workloads across multiple servers. The specification (*Main* agent) given below consists of five agents: a client, the load balancer, and three servers. The client corresponds to a possible environment and is used to simulate various scenarios. The load balancer receives new tasks (*task*) with a private name (*com*), and then interacts with the three servers to know their current load. To do so, a public channel (*e.g.*, *al* for the first server) is used for sending the request and receiving the result. The load balancer compares the different loads and forwards the private name originally submitted by the client to the server with the minimum load. A server has three possible behaviours: it can be asked by the load balancer to return its current load; it can receive a request for a new task (reception of a private name from the load balancer and interaction with the client on this private channel to receive the new load); or it can execute part of its work if its total load is greater than zero. We can see with this simple example how data expressions (natural numbers, comparison, addition, etc.) appear as parameters of agents and channels to specify loads and their manipulation.

$$
\begin{aligned}
&\mathit{Main} = \\
&\quad (\nu\ \mathit{task}, \mathit{al}, \mathit{ar}, \mathit{bl}, \mathit{br}, \mathit{cl}, \mathit{cr})\ (\ \mathit{Client}(\mathit{task}) \mid \mathit{LoadBalancer}(\mathit{task}, \mathit{al}, \mathit{ar}, \mathit{bl}, \mathit{br}, \mathit{cl}, \mathit{cr}) \mid \\
&\quad \mathit{Server}(\mathit{al}, \mathit{ar}, 0\ \mathit{of}\ \mathit{Nat}) \mid \mathit{Server}(\mathit{bl}, \mathit{br}, 0\ \mathit{of}\ \mathit{Nat}) \mid \mathit{Server}(\mathit{cl}, \mathit{cr}, 0\ \mathit{of}\ \mathit{Nat})\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Client}(\mathit{task}) = \\
&\quad (\nu\ \mathit{com}_1)\ \overline{\mathit{task}}\langle \mathit{com}_1\rangle.\overline{\mathit{com}}_1\langle 2\ \mathit{of}\ \mathit{Nat}\rangle.(\nu\ \mathit{com}_2)\ \overline{\mathit{task}}\langle \mathit{com}_2\rangle.\overline{\mathit{com}}_2\langle 1\ \mathit{of}\ \mathit{Nat}\rangle. \\
&\quad (\nu\ \mathit{com}_3)\ \overline{\mathit{task}}\langle \mathit{com}_3\rangle.\overline{\mathit{com}}_3\langle 1\ \mathit{of}\ \mathit{Nat}\rangle.(\nu\ \mathit{com}_4)\ \overline{\mathit{task}}\langle \mathit{com}_4\rangle.\overline{\mathit{com}}_4\langle 2\ \mathit{of}\ \mathit{Nat}\rangle.0
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{LoadBalancer}(\mathit{task}, \mathit{al}, \mathit{ar}, \mathit{bl}, \mathit{br}, \mathit{cl}, \mathit{cr}) = \\
\quad \mathit{task}(\mathit{com}).\ \overline{\mathit{al}}.\mathit{al}(v_1 : \mathit{Nat}).\ \overline{\mathit{bl}}.\mathit{bl}(v_2 : \mathit{Nat}).\ \overline{\mathit{cl}}.\mathit{cl}(v_3 : \mathit{Nat}). \\
\quad\ (\quad [(v_1 \leq v_2)\ \mathit{and}\ (v_1 \leq v_3)]\ \overline{\mathit{ar}}\langle \mathit{com}\rangle.\mathit{LoadBalancer}(\mathit{task}, \mathit{al}, \mathit{ar}, \mathit{bl}, \mathit{br}, \mathit{cl}, \mathit{cr}) \\
\quad\quad + [(v_2 \leq v_1)\ \mathit{and}\ (v_2 \leq v_3)]\ \overline{\mathit{br}}\langle \mathit{com}\rangle.\mathit{LoadBalancer}(\mathit{task}, \mathit{al}, \mathit{ar}, \mathit{bl}, \mathit{br}, \mathit{cl}, \mathit{cr}) \\
\quad\quad + [(v_3 \leq v_1)\ \mathit{and}\ (v_3 \leq v_2)]\ \overline{\mathit{cr}}\langle \mathit{com}\rangle.\mathit{LoadBalancer}(\mathit{task}, \mathit{al}, \mathit{ar}, \mathit{bl}, \mathit{br}, \mathit{cl}, \mathit{cr})\ )
\end{array}
$$

$$
\begin{array}{l}
\mathit{Server}(\mathit{ld}, \mathit{rq}, \mathit{totalload} : \mathit{Nat}) = \\
\quad \mathit{ld}.\overline{\mathit{ld}}\langle \mathit{totalload}\rangle.\mathit{Server}(\mathit{ld}, \mathit{rq}, \mathit{totalload}) \\
\quad + \mathit{rq}(\mathit{req}).\mathit{req}(\mathit{newload} : \mathit{Nat}).\mathit{Server}(\mathit{ld}, \mathit{rq}, \mathit{totalload} + \mathit{newload}) \\
\quad + [\mathit{totalload} > 0]\ \overline{\mathit{execute}}\langle \mathit{ld}, \mathit{totalload}\rangle.\mathit{Server}(\mathit{ld}, \mathit{rq}, \mathit{totalload} - 1)
\end{array}
$$

3 Translation to LNT

Most of the π-calculus constructs are translated quite straightforwardly into LNT because of its high level of expressiveness. Nevertheless, we faced some subtle difficulties in obtaining a translation as succinct as possible while still preserving the LTS semantics, *i.e.*, mapping each transition of a π-calculus agent to a transition of the resulting LNT process. One of the main problems was to encode the binary, unidirectional, and mobile communication of π-calculus into a specification language enabling multi-way and bidirectional communication on static channels.

Since mobile communication cannot be described directly using LNT static channels, we overcome this issue by exploiting the data types and synchronization features of LNT. We represent π-calculus channel names as values of a LNT datatype *Chan*, which defines all the public and private names appearing in the specification. Then, we model channel mobility between π-calculus agents by communicating *Chan* values along LNT channels. Binary unidirectional communications and two-among-n synchronizations, which cannot be directly described in LNT, are encoded by means of dedicated LNT channels (one for each π-calculus parallel composition operator), on which the sender and receiver are indicated explicitly using process identifiers and placeholders. Communication on a π-calculus channel is translated in LNT as a choice on all LNT channels connecting the current agent to its environment. The translation of the original π-calculus to LNT is detailed in [9].

4 Tool Support and Verification with CADP

The translation from our applied π-calculus to LNT has been automated by the translator PIC2LNT 2.0, implemented using the SYNTAX+TRAIAN compiler construction technology [7]. The tool consists of about 2,300 lines of SYNTAX code, 4,800 lines of LOTOS NT code, and 700 lines of C code[1].

Figure 1 gives an overview of the complete tool chain. Given a specification in applied π-calculus, possibly containing data types and functions described in

[1] The version 1.0 of PIC2LNT, which handled the original π-calculus (without data manipulation), consisted of about 3,700 lines of code.

LNT libraries, PIC2LNT translates it into an equivalent LNT specification, which is accepted as input by the CADP tools. The resulting LNT specification is connected by the LNT.OPEN tool (via an intermediate translation to LOTOS) to the OPEN/CÆSAR environment [6], which gives access to all the on-the-fly verification tools of CADP. The `pic2lnt_dyn.tnt` file (static code) contains external C functions for generating fresh channel names and process identifiers.

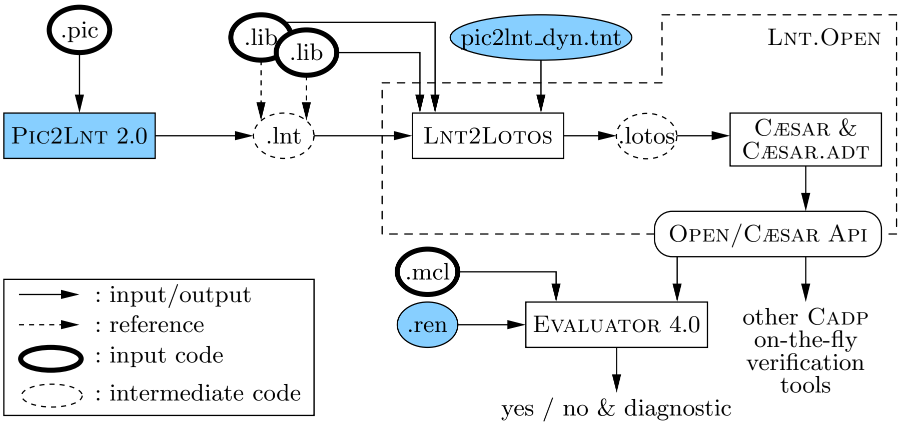


Fig. 1. Overview of the tool chain

As illustrated on Figure 1, one can use the EVALUATOR 4.0 on-the-fly model checker to verify temporal properties specified in MCL [11], an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. MCL is suitable for analyzing applied π-calculus specifications, because the properties can involve both the channel names and the data values transmitted. The LTS actions, which carry additional information introduced during the translation to LNT, are renamed on-the-fly to retrieve the original π-calculus format using a predefined label renaming file.

Going back to the load balancing system specified in Section 2, the LTS of the resulting LNT specification contains 2,007 states and 5,450 transitions. As an example, we can check that this LTS satisfies the MCL data-based response property below, which states that every time a server has begun an execution, it will eventually exhaust its workload by executing it one unit at a time:

$$
\begin{array}{l}
[\mathit{true}^*.\{\mathit{execute}\ ?\mathit{ld}{:}\mathit{String}\ ?\mathit{load}{:}\mathit{Nat}\ ...\ \mathit{where}\ \mathit{load} > 1\}] \\
\quad \mu X(\mathit{crt\_load}{:}\mathit{Nat} := \mathit{load} - 1).(\\
\quad\quad \langle \mathit{true} \rangle\, \mathit{true} \wedge \\
\quad\quad [\neg\{\mathit{execute}\ !\mathit{ld}\ !\mathit{crt\_load}\ ...\}]\ X(\mathit{crt\_load}) \wedge \\
\quad\quad [\{\mathit{execute}\ !\mathit{ld}\ !\mathit{crt\_load}\ ...\ \mathit{where}\ \mathit{crt\_load} > 1\}]\ X(\mathit{crt\_load} - 1) \\
\quad)
\end{array}
$$

The action predicates enclosed between curly braces enable to capture the information present on LTS actions, *i.e.*, the channel names (interpreted as character strings) and the data values transmitted. The box modality matches all sequences that end up, after zero or more steps, with an *execute* action carrying a channel name *ld* and a workload *load*. These values are captured and used later in the parameterized minimal fixed point operator μX, which expresses the inevitable reachability of consecutive *execute* actions that carry decreasing workloads.

The Pic2Lnt 2.0 translator is currently provided as a Cadp plug-in. The manual page and the executable files for several architectures (Mac computers, Pcs running Linux or Windows, Solaris workstations) are available on-line [10].

5 Experimental Evaluation

We applied Pic2Lnt 2.0 on a benchmark of π-calculus specifications, which includes most of the examples provided with Mwb, as well as applied π-calculus examples that we specified ourselves. Our benchmark currently contains 284 files, totalizing about 5,200 lines of π-calculus, which were translated in about 50,000 lines of Lnt. This expansion in size, which is negligible given the speed of the Lnt compiler, is caused partly by the complexity of the translation (one new Lnt channel per parallel composition operator) and partly by the verbosity of Lnt w.r.t. the compact algebraic notation of the π-calculus.

The table below shows a few examples from the Mwb distribution. For each example, the table gives the number of agents, the size of the specification before and after translation, and some quantitative information (size, time) about the Lts generated using Pic2Lnt 2.0 and the Cadp exploration tools.

File name	Description	|Agents|	Nb. of lines .pic	.lnt	Lts $\|S\|/\|T\|$	Time
memcell1	Memory cell	2	7	82	10 / 100	0.39s
memcell2	Memory cell	2	7	91	91 / 910	0.39s
abp-bv	Alternating bit protocol	7	35	257	1,281 / 4,320	1.24s
thandover	Mobile network	6	35	257	11 / 18	0.56s
handstrong	Mobile network	9	40	318	39,909 / 76,679	0.68s
pbool	Boolean operations	6	38	950	4 / 678	1.63s

Our tool support for the applied π-calculus is already used for teaching purposes at the University of Saarbrücken (Germany). It is also currently used for specifying and verifying self-deployment and other self-management protocols designed in the context of the OpenCloudware[2] project, which aims at building an open software engineering platform for the collaborative development of distributed applications to be deployed on multiple cloud infrastructures. Since the applied π-calculus is convenient for specifying many kinds of mobile systems (*e.g.*, Web services, autonomic applications, cloud computing protocols, software architectures, biological systems, cryptographic protocols, etc.), we believe that our tool support can provide a useful service in a wide range of application areas.

[2] `http://opencloudware.org`

6 Concluding Remarks

We introduced in this paper an applied π-calculus equipped with data-handling features, and proposed a translation of this language into the LNT value-passing process algebra. This translation, automated by the PIC2LNT 2.0 tool, enables the analysis of applied π-calculus specifications using all verification tools of CADP. As far as we are aware, this results in one of the few operational frameworks for verifying an applied π-calculus. PROVERIF [2] is an alternative approach focused on the verification of cryptographic protocols and security properties (secrecy, authentication, etc.). In contrast, our solution is independent of any application domain and provides a larger panel of verification techniques.

Acknowledgments. We are grateful to Hubert Garavel for his valuable feedback about the applied π-calculus and the connection of PIC2LNT 2.0 to CADP.

References

1. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. Elsevier (2001)
2. Blanchet, B., Smyth, B.: Proverif: Automatic cryptographic protocol verifier, user manual and tutorial, version 1.86pl3 (2011)
3. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference manual of the LOTOS NT to LOTOS translator (version 5.1). Inria/Vasy, 109 pages (2010)
4. Dam, M.: Model Checking Mobile Processes. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 22–36. Springer, Heidelberg (1993)
5. Ferrari, G., Ferro, G., Gnesi, S., Montanari, U., Pistore, M., Ristori, G.: An Automata Based Verification Environment for Mobile Processes. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 275–289. Springer, Heidelberg (1997)
6. Garavel, H.: OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
7. Garavel, H., Lang, F., Mateescu, R.: Compiler Construction Using LOTOS NT. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 9–13. Springer, Heidelberg (2002)
8. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
9. Mateescu, R., Salaün, G.: Translating Pi-Calculus into LOTOS NT. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 229–244. Springer, Heidelberg (2010)
10. Mateescu, R., Salaün, G.: Pic2Lnt: A translator from the applied pi-calculus to LOTOS NT (2012), http://convecs.inria.fr/software
11. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)

12. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Inf. Comput. 100(1), 1–77 (1992)
13. Montanari, U., Pistore, M.: Checking Bisimilarity for Finitary π-Calculus. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 42–56. Springer, Heidelberg (1995)
14. Victor, B., Moller, F.: The Mobility Workbench – A Tool for the π-Calculus. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 428–440. Springer, Heidelberg (1994)

An Overview of the mCRL2 Toolset and Its Recent Advances

Sjoerd Cranen[1], Jan Friso Groote[1], Jeroen J.A. Keiren[1], Frank P.M. Stappers[1], Erik P. de Vink[1,2], Wieger Wesselink[1], and Tim A.C. Willemse[1]

[1] Eindhoven University of Technology
[2] Centrum Wiskunde & Informatica

Abstract. The analysis of complex distributed systems requires dedicated software tools. The `mCRL2` language and toolset have been developed to support such analysis. We highlight changes and improvements made to the toolset in recent years. On the one hand, these affect the scope of application, which has been broadened with extended support for data structures like infinite sets and functions. On the other hand, considerable progress has been made regarding the performance of our tools for state space generation and model checking, due to improvements in symbolic reduction techniques and due to a shift towards parity game-based solving. We also discuss the software architecture of the toolset, which was well suited to accommodate the above changes, and we address a number of case studies to illustrate the approach.

1 Introduction

Distributed systems and parallel programs are becoming increasingly common as a result of easy access to cheap multi-core processors and the popularity of paradigms such as cloud computing. These systems are notoriously difficult to design correctly. To a large extent this is caused by the concurrency that results in a lack of insight in the global configuration of a system, and the sheer number of different configurations in which a system can be at any moment. Design flaws may result in loss of data or hanging software. Race conditions are a well-known example of such flaws. While an occasional hiccup may be tolerable for non-critical applications, this may be unacceptable if an application causes significant financial losses or increases safety risks.

The `mCRL2` toolset is designed to reason about distributed and concurrent systems. `mCRL2` is based on the process algebra μ`CRL` [7] and inherits its axiomatic view on processes. In μ`CRL`, various methodologies for manually proving correctness of processes based on axiomatic reasoning were developed; these were adopted in `mCRL2`. The `mCRL2` language, like its predecessor, is designed in such a way that it does not restrict the expressive freedom of the user. The data theory is still rooted in the theory of ADTs, but now comes with many built-in data types. Compared to μ`CRL`, the process language has changed slightly but crucially, so semantics can be provided to languages with a notion of *true concurrency*.

The introduction of *parameterised boolean equation systems* [23] in the mCRL2 toolset clearly marks the transition to a verification paradigm based on model checking. The model checking approach complements the axiomatic verification methodology offered in the toolset. Currently, the mCRL2 toolset consists of over 60 tools that together allow visualisation, simulation, minimisation and model checking of complex systems. This paper aims to offer an overview of the toolset and its usage. We highlight its conceptual and technical essentials, of which we illustrate the domain of application, emphasising on recent developments.

First, we provide a cursory overview of the mCRL2 language. We then explain the notions of *linear process* and *equation system*, which play a fundamental role in many of the algorithms implemented in the mCRL2 toolset. The most recent improvements and additions are highlighted, addressing amongst others tool performance, support for analysing real-time systems, and solving equation systems via parity games. To broaden the scope of application, mCRL2 interfaces with other specification languages. We report on initial investigations to reduce the work needed to keep these interfaces up-to-date.

As the code base of the mCRL2 toolset has expanded substantially over the last few years, maintainability has become an important aspect in the development of the toolset. We describe our efforts to reduce the amount of hand-written code, and to improve readability and documentation of our software. These and other concerns, such as interoperability, have led to architectural changes that we mention briefly.

The uses of the language and tools are sketched by summarising a selection of illustrative case studies conducted with mCRL2. We indicate where recently added techniques were instrumental for these case studies. Finally, we position our toolset in the broader context of verification tooling, and give an outlook on the challenges ahead.

Documentation, sources and binaries of the mCRL2 toolset can be downloaded from the mCRL2 website www.mcrl2.org. The toolset is open source; the associated boost license allows free use for any purpose. A user manual also containing a tutorial can be found in the user documentation section of the website. The tutorial introduces the reader to the basic concepts and syntax and provides guidance for the tools most commonly used. Lecture notes used for a master course at Eindhoven University of Technology and Delft University of Technology, approaching a final draft, are available from the mCRL2 website too.

2 mCRL2: Approach, Applications and Challenges

The mCRL2 language consists of three different sublanguages: a data language, a process language, and a property language. Following the philosophy underlying mCRL2, convenience of modelling and expressiveness have been leading in the respective definitions. We briefly discuss the three sublanguages below. For an in-depth treatment of the language, we refer to the website and the publications and material mentioned there.

In mCRL2 data and transformations on data are described using abstract data types. This allows users to create their own data types by defining the appropriate constructors and by providing functions operating on the data types. The mCRL2 data language has built-in support for commonly used data types, like the booleans, natural numbers, integers and reals. The usual operations on these data structures are predefined. Complex types can be constructed using type constructors such as sets, lists, and functions over any data type. Notation for built-in data types stays close to mathematics: numbers are written as sequences of decimals, without a limit on the size of the numbers. Sets are written using set comprehension. Functions are first-class citizens, and can be used to obtain concise models. The language allows in-line lambda abstraction as well as function updates. For example, the function doubling every natural number can be defined using the lambda abstraction `lambda n:Nat.n+n`. The function that doubles every natural number, but maps the number 4 to 0 can be defined using a function update `(lambda n:Nat.n+n)[4->0]`.

The behaviour of a system is described by processes, composed from a set of user-defined actions and a set of operators on actions and processes. These operators include multi-action composition, sequential, alternative and parallel composition and abstraction operators. The language also offers primitives to model real-time systems. Processes are defined in the context of data definitions describing the data types that are used and the operations upon them. This permits the modelling of systems whose behaviour crucially depends on the data that is exchanged: actions can be parameterised by data and *if-then-else* constructs allow for specifying conditional process behaviour. The semantics of processes is defined using a structural operational semantics, which associates with every expression in the language a *labelled transition system* (LTS). Such a labelled transition system is viewed as a graph consisting of vertices and edges, where each edge is labelled with an action, which in turn can have data parameters. The information contained in vertices is represented by a process expression and a valuation of its data parameters, but is unobservable; behaviour is determined by the actions.

High-level properties can be described using an extension of Kozen's propositional modal μ-calculus. Least and greatest fixpoint operators, which may be nested arbitrarily, can be used in combination with modal operators to describe requirements of increasing complexity. In this manner it is for instance possible to specify fairness properties, thereby staying true to the design philosophy that the modeller should not be restricted in his or her expressive freedom. The property language is equipped with constructs for reasoning about timed processes. Semantically, expressions in the property language identify a set of states in a given labelled transition system (namely, those states that satisfy the property).

Although unrestrictive, the μ-calculus is an intricate formalism. Its usability is improved by providing a set of powerful, intuitive macros, inspired by the regular expressions found in PDL. In many practical situations, this eliminates the need for fixpoint operators. For instance, safety properties asserting that a system should not exhibit a sequence of actions matching the regular expression r simply becomes `[`r`]false`. The existence of such a sequence is expressed as `<`r`>true`. By mixing regular expressions and fixpoints, one can build more complex formulae that are still easy to read. For instance, the expression `nu X.<`r`>X` asserts that there is an infinite path of action sequences matching the regular expression r. If, for instance, $r =$ `a*.b`, it says that there is a path consisting of an infinite number of `b` actions, interrupted by finite sequences of `a` actions.

The ability to use parameterised actions in the process specification language requires similar capabilities in the property language. Like processes, properties are therefore interpreted in the context of a data specification. Fixpoint variables and actions can be parameterised with data, boolean expressions may contain data variables, and universal and existential quantification over (possibly infinite) data types are allowed. Action formulae denote (potentially infinite) sets of parameterised actions. For example, one may write `true` to denote the set of all actions, or `exists n:Nat.val(n > 5)&& s(n)` to denote the set of `s(n)` actions, where `n>5`. The property `[true*. exists n: Nat.val(n > 5) && s(n)]false` then expresses that such an action never occurs.

The expressiveness of the `mCRL2` property language makes it well-suited for reasoning about complex distributed systems. Its expressivity is witnessed by the fact that one can easily encode the counting μ-calculus [26] in it, which is known to be strictly more expressive than the propositional μ-calculus. The incorporation of data even enables succinct transformations from popular temporal logics. In [12], we reported on a linear transformation from `CTL*` to our μ-calculus; the transformation of `CTL*` to the equational propositional modal μ-calculus is exponential [5].

The expressive power of the `mCRL2` language also has serious consequences as far as automation is concerned. Heuristics are required to work around the general undecidability of the data theory. Quantifier elimination cannot simply rely on exhaustive enumeration of all elements of a data type in case the carrier of the latter is of infinite size. The ability to use unrestricted mixing of least and greatest fixpoints in the μ-calculus may lead to computationally intractable decision problems. In the past years, we have made significant improvements in the `mCRL2` toolset to cope with the consequences of the expressive power of the `mCRL2` language.

3 The mCRL2 Toolset

The `mCRL2` toolset consists of over 60 tools that together allow for analysing complex system designs formally described in the `mCRL2` language. Internally, the toolset relies on two types of objects, viz. *linear processes* [21] and *parameterised boolean equation systems* [23]. The toolset offers full control over these objects, equipping users with tools to manipulate and transform them. Below, we explain these concepts in more detail, and we indicate what progress was made in recent years.

Linear Processes. Any analysis on `mCRL2` specifications is preceded by an automated transformation of the specification to the linear process format.

Technically, a linear process is again an mCRL2 process specification adhering to a restricted grammar, which essentially is a syntactic format for the single-step transition relation that a process induces. That is, a linear process is a recursive equation, in the untimed setting, of the following form:

$$P(d{:}D) = \sum_{i \in I} \sum_{e_i : D_i} c_i(d, e_i) \ \rightarrow \ \alpha_i(d, e_i) \cdot P(f_i(d, e_i))$$

The state space is represented by variable d of sort D. In practice, this is a vector of variables of complex sorts. Each $i \in I$ describes a *condition-action-effect* expression, stating that a multi-action α_i, consisting of actions with parameters that depend on variable d and local variable e_i, can be executed, provided boolean condition c_i evaluates to true for the values for d and e_i. The result of executing this multi-action is a state transition to $f_i(d, e_i)$. The choice between the different condition-action-effect expressions from I is resolved non-deterministically. The transformation to the linear process format is based on the expansion laws of the parallel operator of the mCRL2 process specification language. User control over linear processes is one of the distinguishing advantages of the mCRL2 toolset.

Behaviour-preserving transformations on linear processes are useful for reducing their complexity by either reducing the complexity of the data types occurring in a linear process, reducing the number of data parameters of a process, or by replacing data expressions with simpler ones. In some instances these techniques even allow one to handle processes with infinite state spaces. Typical situations in which such manipulations are very effective occur when verifying data transfer protocols, where the payload of messages is not important.

More recently, an experimental tool was developed to transform linear processes with real-valued data sorts, representing infinite state spaces such as timed systems, into linear processes representing finite ones. The tool performs a form of predicate abstraction, where the predicates are limited to linear equations over the real-valued parameters of the process.

Linear processes can be simulated, and their state space can be explicitly generated and stored. State space generation from a linear process is sped up considerably by caching the evaluation of summands in the spirit of [6], and by pruning parts of the linear process that do not contribute transitions. Typically these techniques speed up state space exploration by a factor 10 to 100. Explicit state spaces can be reduced using behavioural equivalences like strong and branching bisimulation. Implementations of simulation preorders and equivalences, as well as a divergence preserving variant of branching bisimulation have also been made available. Moreover, LTSs can be analysed using a variety of advanced, interactive visualisation techniques for both small and large state spaces in 2D and 3D [24,39].

Parameterised Boolean Equation Systems (PBESs) or just equation systems, for short, are essentially systems of least and greatest fixpoint equations over predicates involving parameterised predicate variables. Typically, a single equation has the form $\mu X(d{:}D) = \varphi$ or $\nu X(d{:}D) = \varphi$. Here, X is a predicate variable,

d is a formal variable of some sort D, and φ is a predicate formula in positive form, containing boolean expressions, predicate variables, conjunctions, disjunctions and existential and universal quantifications. The μ and ν sign indicate whether, respectively, the least or largest solution for X satisfying the equation is desired. Thus, an equation system is viewed as a finite, ordered sequence of equations for distinct predicate variables.

The problem of deciding whether a given property expressed in the μ-calculus holds for a given process specification is automatically encoded in an equation system such that the property holds for the specification if and only if the solution to the equation system is true [22]. Apart from model checking problems, also the equivalence of two processes modulo a process equivalence can be decided by encoding it into an equation system, following the encoding of [9]. This transformation is interesting when comparing infinite state spaces. Comparing finite state spaces is more efficient using traditional algorithms.

We are primarily interested in the solution of a PBES, as it is also the answer to the encoded problem. In many cases, however, manipulations and simplifications are needed before the equation system can actually be solved within the available memory and time. In the past years, we have added new tools implementing solution-preserving manipulations. Inspired by a similar technique operating on linear processes, an algorithm has been added that removes data parameters from propositional variables if they do not affect the solution, see [34]. Other tools implement the automated detection of invariants of equation systems [35] and use these to simplify the predicates in the equations, again without affecting the solution to the encoded verification problem. The computational complexity of these techniques is low, operating at the level of the syntax, but their effects on the time needed to solve the equation systems can be tremendous. Recently, abstract interpretation technology for equation systems was added, allowing one to reduce complex, potentially infinite data types to simpler, finite data types. A recent theoretical analysis of the underlying theory [15] revealed that this technique is more powerful than model checking based on abstractions using modal transition systems, such as [38] and their generalisations using hyper-transitions, see e.g. [42].

Solving a PBES typically proceeds by transforming it into an equation system in which all data parameters and data expressions have been eliminated [36]. Such equation systems, which are systems of fixpoint equations over propositions, are called boolean equation systems or BESs [31]. Solving boolean equation systems is known to be a decidable problem. The transformation process bears many similarities to the computation of a state space from a specification. An essential step in transforming equation systems to boolean equation systems is the simplification of predicates. Quantifier elimination technology is essential to make such transformations efficient. The approach taken here is that of constructor induction, as outlined in [36], which works regardless of whether data types are finite or infinite. Special rules, such as the one-point rule, help speeding up the quantifier elimination, and are often necessary to ensure termination.

An intuitive method for solving boolean equation systems is through Gauss elimination [31]. The algorithms for solving boolean equation systems that were first offered in the toolset are based on this algorithm. While, technically, Gauss elimination is independent of the alternation depth, in practice, this method scaled poorly on verification problems obtained from fairness problems, which require μ-calculus formulae of alternation depth 2 or more. We therefore exploit the tight connection between boolean equation systems and parity games [17,20]. To efficiently generate a parity game from a PBES, an alternative way of generating a PBES from an LPS and a μ-calculus formula was recently introduced. Several algorithms for solving parity games have been made available to users of the toolset. Most notably, implementations of the *Small Progress Measures* [27] algorithm and the *Recursive Algorithm* [50] are available. For most model checking problems, these are very competitive, even for μ-calculus formulae of alternation depth 2 and beyond. Moreover, bisimulation-inspired reductions for boolean equation systems [29] and parity games [14] have been instrumental in solving PBESs where more direct approaches failed.

4 Interfacing with Other Languages

The state space exploration facilities and model checking capabilities of the mCRL2 toolset can be used in combination with various other specification languages.

So-called narration and annotation of security protocols can be expressed in the process algebra LySA, a variant of the π-calculus that uses pattern matching to deal with encrypted data, cf. [8]. Static analysis of LySa processes has been applied to find authenticity and authentication issues. The conversion of a LySa specification into mCRL2, which in particular reflects the treatment of data, makes it possible to do complementary behaviour-oriented analysis.

Using the channel-based coordination language Reo, so-called connectors can be defined to orchestrate the interaction in a component-based system or a service-oriented application [1]. A transformation of Reo connectors into mCRL2 adds model checking to the extensive tool suite for Reo. The synchronicity of ports that is typical for Reo fits well with the notion of multi-action incorporated in mCRL2 and lies at the heart of the efficiency of the transformation.

The mCRL2 toolkit accepts a number of other languages for input. These include the Petri net mark-up language PNML [48], the discretely timed part of the hybrid process algebra χ [3], a subset of executable UML [32], as well as a number of domain specific languages like SML, a control language based on finite state machines used at CERN [18], and TRECS, a language that manages resource availability [33] in the wafer steppers manufactured by ASML.

Not only the many differences between these languages, but also the evolution of their syntax and their semantics makes it difficult to maintain the dedicated tools that implement the various transformations. In fact, some of the front-ends mentioned have been marked deprecated in the latest releases of the mCRL2 toolset. To alleviate part of the burden, we are investigating a generic method to transform external specification formalisms into mCRL2 using Plotkin's structural operational semantics (SOS) as a common representation format.

Using this method, specifications in any language with a structural operational semantics can be transformed into a linear process. This is done by transforming the SOS into an mCRL2 data specification, and the specification under study into an mCRL2 data structure, which are then embedded in a process. This results in an mCRL2 process that encodes the semantics of the specification, and that can be analysed with all the means provided by the mCRL2 toolset. In [44], the underlying algorithm is explained for rules in the De Simone format [16], which is one of the most elementary rule formats for SOS. Extensions to the rule format, e.g. to include predicates, look-aheads and negative premises, can be handled in a similar manner [43].

While the approach is promising from a maintenance point of view, the encoding described above yields models that currently require too much time to verify in practice. Further research is therefore needed to make the technique usable on a larger scale.

5 Architecture and Implementation

The mCRL2 toolset is a collection of tools written in portable C++. Development started around eight years ago, and the code base has steadily grown since then. At present it has more than 200K lines of code, is open source, is supported on 32-bit and 64-bit platforms and runs on most popular operating systems, including Linux, FreeBSD, Windows and Apple Mac OS X. Over the years development and testing of the mCRL2 toolset has matured. The code has been refactored and set up as a collection of libraries with well-defined interfaces. Code has been documented, and regression and performance tests are now run on a daily basis. Recently, commercial spin-off activities based on the mCRL2 toolset have started.

The toolset accommodates two kinds of users. End-users use the toolset for verification and validation of formal models, while the toolset also serves as a vehicle for experimental research. For end-users, correctness of the code and high-performance are the most important. Experimental researchers on the other hand require a high degree of flexibility, since they frequently want to test new ideas and algorithms. Many algorithms have been (re-)written to make the code correspond closely to pseudo-code specifications of the algorithms. This greatly improves the communication between experimental researchers and developers, which is often challenging in academic environments. The pseudo-code is also instrumental in establishing correctness of the algorithms, and in localising bugs.

A number of techniques are employed to support these different kinds of usage. Generic programming is applied to improve adaptability of the code. Notably, a universal framework for traversing the tree-like data structures in mCRL2 has been developed, which lies at the heart of many algorithms in the toolset. This framework uses static polymorphism, both for efficiency reasons and to support a modular design. Code generation from concise specifications makes it easier to incorporate changes, and increases code reuse, which in turn reduces errors.

Most of the traversal framework, and many classes and their operations consist of generated code. Currently about 17% of the code is generated, and this number is expected to increase further.

The mCRL2 toolset has a highly expressive input language. Therefore, test coverage has always been a problem. Recently, random testing has been applied to increase coverage. Randomly generated PBESs have proven to be successful in discovering otherwise hard to find bugs, like subtle cases where name clashes between quantifier variables in formulas were handled incorrectly. Currently the random generation of LPSs and state spaces is under development.

In the backend, mCRL2 provides interfaces to other tools. On the one hand, standardised file formats such as Aldebaran (.aut) and Binary Coded Graphs (.bcg) are used to export labelled transition systems to other tools such as CADP [19]. In the mCRL2 toolset, stable interfaces are provided for state space exploration. These have been designed in such a way that compile and link dependencies of tools using an interface can be kept to a bare minimum, to prevent API breakage. In close collaboration with its developers a coupling has been established with LTSmin [6], that enables symbolic and parallel state space generation of LPSs. Recently, an interface has also been added that enables instantiation of equation systems into parity games using LTSmin. As a result, the parallel and symbolic exploration techniques from LTSmin can now also be used to solve PBESs.

6 Applications and Case Studies

The purpose of the mCRL2 toolset is twofold. On the one hand, it aims to provide a set of state-of-the-art tools for the analysis of distributed systems. On the other hand, it serves as a platform to test research ideas in practice.

Below, we briefly report on three case studies conducted using the toolset, to offer a glimpse into the application domains of mCRL2. The first case study illustrates that the recent integration with LTSmin tool can help to reduce verification times substantially. The second case study illustrates that the mCRL2 multi-action can be essential for modelling systems and that parity game reduction techniques can be crucial for conducting the verification. The third case study demonstrates that case studies can be instrumental in improving the quality of the toolset.

DIRAC: a distributed community grid solution. The high-energy experiments conducted at the large hadron collider of CERN generate a massive amount of raw data. A computing grid solution called DIRAC offers users uniform and reliable access to storage and computing resources. Despite a decade of continuous investment in developing and maintaining DIRAC, parts of the system occasionally enter inconsistent states, leading to a loss of efficiency and a potential loss of data. In an effort to tackle the problem at its root, the critical DIRAC subsystems have been modelled and analysed in mCRL2 [40]. The models of the subsystems were verified using model checking. Modal μ-calculus formulae expressing liveness and safety requirements were formalised. Typical requirements stated, for example, that jobs are always processed once submitted, and that jobs never enter an inconsistent state. Violations of these requirements revealed livelocks and race conditions, explaining phenomena observed in the actual system.

The technology enabling the verification was the symbolic exploration (using the equation system interface with LTSmin, see [28]) and solving of the equation systems encoding the model checking problems. This allowed for a full verification of the system in under 60 seconds on a 64 bit Intel Core Duo (1.6GHz) machine with 2 GB RAM. For comparison, the model checking problem for a single property required more than 50 hours when conducted using explicit state space generation approaches, exploring well over $1.5 \cdot 10^8$ states. Attempts to employ compositional verification, relying on equivalence reductions to minimise state spaces, failed due to the fact that the individual processes that make up the subsystems have infinite state spaces.

FlexRay is a communication protocol that was developed by a consortium of automotive companies. Its final version was published in 2012. The protocol is designed to provide a reliable, high-bandwidth communication channel between nodes, with predictable timing properties. The protocol is *time-triggered*, that is, the protocol relies on nodes (senders and receivers of messages) to have synchronised clocks, and operates by allocating bandwidth to senders based on a global, cyclic schedule. Using mCRL2, the FlexRay startup procedure, which ensures that activated nodes will find each other and will correctly initialise their local view on the global schedule, was modelled and checked for correctness [11]. The rich data language, and the modularity of the process language of mCRL2 allowed to specify the FlexRay protocol closely. In the protocol, there is a notion of *macroticks*, clock ticks that are generated by one process and communicated to the other processes using events. To model the synchronisation that these macroticks induce, multi-actions were used to create a form of barrier synchronisation.

To review the robustness of the protocol, faults that might occur in the system were modelled, which could mostly be done by making small, local changes to the fault-free model. The property language of mCRL2 showed itself conveniently expressive to define relatively complicated properties. For instance, the property that eventually all nodes in the network will keep sending messages according to their schedule was expressed as a μ-calculus formula that uses fixpoints parameterised with data variables representing sets, and user-defined functions to specify the schedule. The properties were verified by creating a PBES, expanding it and solving the resulting BES. Solving time for these (large) equation systems was reduced by interpreting the BES as a parity game, reducing that game using a notion of stuttering equivalence tailored to parity games, and then solving the reduced game [13].

Domain Specific Languages. Domain specific languages, or DSLs, have become increasingly popular with high-tech industry to speed up their design and development cycles. Although DSLs provide an easy way to design software for a specific domain, they do not guarantee correctness of the designs. Through DSLs, however, techniques from the mCRL2 toolset can be made available to industry.

If an SOS-style operational semantics is available for a DSL, the transformation technique discussed in Section 4 can be used to analyse it using mCRL2. However, many domain specific languages are still defined informally. In [45] we report on a case study of the formalisation of an industrial DSL, called TRECS. The execution semantics was implicitly defined by the implementation of the TRECS interpreter. By formalising the language, that is, by creating an SOS for its syntactic constructs and subsequent application of the semantic transformation, we were able to discover—and improve upon—sub-optimal design decisions using the mCRL2 toolset.

To further investigate the applicability of the approach, we took the formal definition of the mCRL2 language itself and encoded it into mCRL2 again by applying the same procedure [46]. The SOS consisted of 43 deduction rules and resulted in an mCRL2 specification of slightly over 1000 lines of code. Our effort revealed a number of subtle differences between the specified, intended and the implemented semantics. In particular, the definition of the mCRL2 language allows for the use of existential quantifiers within a set comprehension scheme, but this possibility was overlooked in the actual implementation of the linear specification generator. The exercise led to improvements in the toolset and the documentation of the language.

7 Related Work

The mCRL2 toolset was originally based on the toolset associated with μCRL [7]. As such, a lot of the functionality of the μCRL toolset can still be found in the mCRL2 toolset.

The toolset that—in terms of functionality—most resembles the mCRL2 toolset is CADP, developed in Grenoble [19]. It uses the specification language Lotos NT, which, like the process language of mCRL2, has its roots in process algebra; it has a property language that is, like the mCRL2 property language, based on a variant of the propositional μ-calculus, and, like in the mCRL2 toolkit, verification is conducted using equation systems. Both toolsets offer the basic functionality of minimising explicit labelled transition systems and visualising these; CADP offers a slightly richer set of equivalences that can be used to reduce with, whereas mCRL2 offers more advanced interactive 2D and 3D visualisation tooling. There are a few key differences between the two toolsets. While the mCRL2 toolset is fully open source, CADP's license imposes more restrictions. Model checking in CADP is essentially limited to alternation-free μ-calculus formulae, with limited support for alternation depth 2 formulae, whereas potentially mCRL2 can verify μ-calculus formulae of arbitrary alternation depth. Unlike CADP, mCRL2 can be used to specify and analyse real-time systems. On the other hand, CADP provides features to support performance evaluation, which are lacking in mCRL2. Finally, there are differences in the philosophy between CADP and mCRL2: the latter provides full control over objects such as linear processes and equation systems, whereas in CADP objects fulfilling similar roles are hidden from the user.

Process algebras from the CSP family are less closely related to mCRL2. For example, the FDR2 toolset [41] is based on checking refinement relations such as failure-divergence inclusion between specifications and implementations. It has support for static analysis and compositional reasoning; facilities for model checking are limited to a predefined set of properties such as (the absence of) livelock, deadlock and determinism. The PAT toolset [47] provides similar features, but additionally supports specifying and analysing real-time systems and it is capable of LTL-based model checking. Furthermore, it comes with advanced techniques such as partial order reduction and symmetry reduction.

Prominent tools focussing on model checking include SPIN [25] and nuSMV [10]. The languages supported by these tools have more restricted data types (generally booleans or bits, limited range integers and finite arrays). SPIN uses a C-like process specification language Promela for the analysis of parallel programs. It primarily focusses on LTL model checking. Properties can be established by augmenting the specification with assertions and so-called 'never claims', which are either obtained from LTL formulae or constructed manually. The tool is most famous for its use of partial order reduction and bit hashing technology. The tool DiVinE is an LTL model checker built for grid and multi-core platforms [2]. It is an automaton-based tool providing a high-performance parallel computing engine. The nuSMV toolset exploits clever data structures such as BDDs to compactly represent large state spaces. Model checking in nuSMV is currently limited to CTL and LTL properties. It also offers support for bounded model checking using SAT solving.

Several toolsets are optimised for verifying specifications with predominantly quantitative aspects. These include real-time and probabilistic model checking, with tools such as Uppaal [4] and Prism [30] The tool Uppaal is based on the notion of timed automata and uses graphs to draw behaviour which can be used to describe timed behaviour. Model checking of a restricted temporal logic is solved elegantly relying on efficient representations and manipulations of time regions. The tool Prism targets discrete and continuous-time Markov chains and decision processes. It supports simulation and model checking of PCTL and CSL.

In Section 5, we already mentioned the LTSmin toolset [6] as one of the back-ends for mCRL2. Contrary to the toolsets listed above, LTSmin has no dedicated language. Instead, it provides highly optimised state space generation tools employing multi-core, parallel and symbolic reachability analysers and model checkers, and it is used as back-ends for, e.g., DiVinE, SPIN and mCRL2.

8 Closing Remarks

The mCRL2 language and toolset provide end-users with state-of-the-art tools for analysing complex, distributed systems. In developing the mCRL2 toolset we aim to uphold a consistent and reliable user experience across the various supported operating systems, viz., Linux, Windows, Apple Mac OS X and FreeBSD. For instance, we recently ported all our graphical tools from wxWidgets to Qt for this reason. On the other hand, the toolset serves as a platform for testing research ideas in practice. This requires flexible code that is easy to adapt. Some of the older parts of the toolset have not been written with adaptability in mind, making it harder to experiment with these. Efforts are being made to change this. For example, the type checker of the language is scheduled for replacement by a much more generic and modularised version. While we consider such maintenance to be necessary for the progress of the toolset, it distracts from more fundamental research.

Several challenges lie ahead. Underlying many of the algorithms for manipulating linear processes and equation systems in the `mCRL2` toolset is a rewrite engine. The rewriter enables automated reasoning about data expressions found in the linear processes and equation systems. Therefore, the efficiency of our tools depends, to a large extent, on the performance of the rewriter. Currently, we use just-in-time rewriting [37], which has been improved using strategy trees and matching trees [49]. These are in essence techniques that reduce the number of checks that have to be done in the rewrite engine. Nonetheless, the current first-order rewriter sometimes causes performance problems when dealing with more advanced language constructs such as lambda expressions, which we expect to be able to solve using a generic higher-order rewriter. Such a rewrite engine is currently under development.

At the same time, a few of our algorithms rely on a theorem prover based on binary decision diagrams with equations. It may be beneficial to use dedicated provers like SMT solvers for some problems instead. Limited support for integrating SMT solvers is already present in several experimental tools. Integrating them more robustly in the toolset and using them in more places is part of our ongoing investigations. In particular, we are investigating possible ways to connect SMT solvers with the abstraction tooling for `PBES`s [15].

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
2. Barnat, J., Brim, L., Ročkai, P.: DiVinE Multi-Core – A Parallel LTL Model-Checker. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 234–239. Springer, Heidelberg (2008)
3. van Beek, D.A., et al.: Syntax and consistent equation semantics of hybrid Chi. Journal of Logic and Algebraic Programming 68(1-2), 129–210 (2006)
4. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
5. Bhat, G., Cleaveland, R.: Efficient model checking via the equational μ-calculus. In: LICS, pp. 304–312. IEEE Computer Society (1996)
6. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
7. Blom, S., Fokkink, W., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: μCRL: A Toolset for Analysing Algebraic Specifications. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Heidelberg (2001)
8. Bodei, C., et al.: Automatic validation of protocol narration. In: CSFW 2003, pp. 126–140. IEEE (2003)
9. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 120–135. Springer, Heidelberg (2007)

10. 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)
11. Cranen, S.: Model Checking the FlexRay Startup Phase. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 131–145. Springer, Heidelberg (2012)
12. Cranen, S., Groote, J.F., Reniers, M.A.: A linear translation from CTL* to the first-order modal μ-calculus. Theoretical Computer Science 412, 3129–3139 (2011)
13. Cranen, S., Keiren, J.J.A., Willemse, T.A.C.: Stuttering Mostly Speeds Up Solving Parity Games. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 207–221. Springer, Heidelberg (2011)
14. Cranen, S., Keiren, J.J.A., Willemse, T.A.C.: A Cure for Stuttering Parity Games. In: Roychoudhury, A., D'Souza, M. (eds.) ICTAC 2012. LNCS, vol. 7521, pp. 198–212. Springer, Heidelberg (2012)
15. Cranen, S., Gazda, M.W., Wesselink, J.W., Willemse, T.A.C.: Abstraction in parameterised boolean equation systems. Technical Report 13-01, Eindhoven University of Technology (2013)
16. De Simone, R.: Higher-level synchronising devices in Meije-SCCS. Theoretical Computer Science 37, 245–267 (1985)
17. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: FOCS 1991, pp. 368–377. IEEE (1991)
18. Franek, B., Gaspar, C.: SMI++ object-oriented framework for designing and implementing distributed control systems. IEEE Transactions on Nuclear Science 52(4), 891–895 (2005)
19. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
20. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
21. Groote, J.F., Ponse, A., Usenko, Y.S.: Linearization in parallel pCRL. Journal of Logic and Algebraic Programming 48(1-2), 39–70 (2001)
22. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. Science of Computer Programming 56(3), 251–273 (2005)
23. Groote, J.F., Willemse, T.A.C.: Parameterised Boolean equation systems. Theoretical Computer Science 343(3), 332–369 (2005)
24. van Ham, F., van de Wetering, H., van Wijk, J.J.: Visualization of state transition graphs. In: Andrews, K., et al. (eds.) INFOVIS 2001, pp. 59–63 (2001)
25. Holzmann, G.J.: The SPIN model checker: Primer and reference manual. Addison-Wesley (2003)
26. Janin, D., Lenzi, G.: Relating levels of the mu-calculus hierarchy and levels of the monadic hierarchy. In: LICS 2001, pp. 347–356. IEEE (2001)
27. 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)
28. Kant, G., van de Pol, J.: Efficient instantiation of parameterised boolean equation systems to parity games. In: Proc. GRAPHITE 2012. EPTCS, vol. 99, pp. 50–65 (2012)
29. Keiren, J.J.A., Willemse, T.A.C.: Bisimulation Minimisations for Boolean Equation Systems. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 102–116. Springer, Heidelberg (2011)

30. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 52–66. Springer, Heidelberg (2002)
31. Mader, A.: Verification of Modal Properties Using Boolean Equation Systems. PhD thesis, Technische Universität München (1997)
32. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesly (2002)
33. van den Nieuwelaar, N.J.M.: Supervisory Machine Control by Predictive-reactive Scheduling. PhD thesis, Eindhoven University of Technology (2004)
34. Orzan, S., Wesselink, W., Willemse, T.A.C.: Static Analysis Techniques for Parameterised Boolean Equation Systems. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 230–245. Springer, Heidelberg (2009)
35. Orzan, S.M., Willemse, T.A.C.: Invariants for parameterised Boolean equation systems. Theoretical Computer Science 411(11-13), 1338–1371 (2010)
36. Ploeger, B., Wesselink, W., Willemse, T.A.C.: Verification of reactive systems via instantiation of parameterised Boolean equation systems. Information and Computation 209(4), 637–663 (2011)
37. van de Pol, J.: JITty: A Rewriter with Strategy Annotations. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 367–370. Springer, Heidelberg (2002)
38. van de Pol, J., Espada, M.V.: Modal Abstractions in μCRL. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 409–425. Springer, Heidelberg (2004)
39. Pretorius, A.J., van Wijk, J.J.: Bridging the semantic gap: Visualizing transition graphs with user-defined diagrams. IEEE Computer Graphics and Applications 27, 58–66 (2007)
40. Remenska, D., Willemse, T.A.C., Verstoep, K., Fokkink, W., Templon, J., Bal, H.: Using model checking to analyze the system behavior of the LHC production grid. In: CCGrid 2012, pp. 335–343. IEEE (2012)
41. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1998)
42. Shoham, S., Grumberg, O.: 3-valued abstraction: more precision at less cost. Information and Computation 206(11), 1313–1333 (2008)
43. Stappers, F.P.M.: Bridging Formal Models: An Engineering Perspective. PhD thesis, Eindhoven University of Technology (2012)
44. Stappers, F.P.M., Reniers, M.A., Weber, S.: Transforming SOS Specifications to Linear Processes. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 196–211. Springer, Heidelberg (2011)
45. Stappers, F.P.M., Weber, S., Reniers, M.A., Andova, S., Nagy, I.: Formalizing a Domain Specific Language Using SOS: An Industrial Case Study. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 223–242. Springer, Heidelberg (2012)
46. Stappers, F.P.M., Reniers, M.A., Groote, J.F., Weber, S.: Dogfooding the formal semantics of mCRL2. In: Bowen, J., Zhu, H. (eds.) 35th SEW. IEEE (2012) (to appear)
47. 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)
48. Weber, M., Kindler, E.: The Petri Net Markup Language. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 124–144. Springer, Heidelberg (2003)
49. van de Weerdenburg, M.: Efficient Rewriting Techniques. PhD thesis, Eindhoven University of Technology (2009)
50. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200(1-2), 135–183 (1998)

Analysis of Boolean Programs

Patrice Godefroid[1] and Mihalis Yannakakis[2]

[1] Microsoft Research
pg@microsoft.com
[2] Columbia University
mihalis@cs.columbia.edu

Abstract. Boolean programs are a popular abstract domain for static-analysis-based software model checking. Yet little is known about the complexity of model checking for this model of computation. This paper aims to fill this void by providing a comprehensive study of the worst-case complexity of several basic analyses of Boolean programs, including reachability analysis, cycle detection, LTL, CTL, and CTL* model checking. We present algorithms for these problems and show that our algorithms are all *optimal* by providing matching lower bounds. We also identify particular classes of Boolean programs which are easier to analyse, and compare our results to prior work on pushdown model checking.

1 Introduction

Boolean programs are programs in which all variables have Boolean type and which can contain recursive procedures. They are a popular abstract domain for static-analysis-based software model checking, pioneered by the SLAM project [5]. SLAM verifies control-flow dominated properties of Windows device drivers by abstracting a C program with a Boolean program generated using *predicate abstraction* (e.g., [21]). The Boolean program contains the same procedures and control flow as the original program, but uses Boolean variables to keep track of the values of predicates over variables of the original program, abstracting its "data part". The level of abstraction can be adjusted iteratively and automatically by changing the finite set of predicates being tracked, using a process sometimes called "Counter-Example Guided Abstraction Refinement" (CEGAR). Since SLAM, other tools have adopted Boolean programs as an abstract domain for software model checking, such as BLAST [23], YASM [22], TERMINATOR [15] and YOGI [19].

The main advantage of Boolean programs compared to finite-state transition systems is that their stack allows a *precise* representation of procedure calls, including recursion, while providing a model of computation for which many interesting properties are still *decidable*. Indeed, Boolean programs have the same expressiveness as pushdown systems [4], for which many properties of interest, such as reachability and temporal-logic model checking, are decidable [8], even though their set of reachable states can be infinite.

Several algorithms for reachability analysis of Boolean programs have been proposed in the literature. For instance, [4] discusses a symbolic model checker for safety properties (reachability analysis) using BDDs as procedure summaries. [17] extends the previous results to Linear Temporal Logic (LTL) model checking, which can also check liveness properties with fairness constraints. [25] discusses how to reduce reachability analysis of Boolean programs to SAT solving. More recently, [7] investigates how to use SAT encodings, instead of BDDs, to represent procedures summaries and to use a QBF solver for reachability analysis.

Yet, despite this prior work, little is known about the complexity of model checking for Boolean programs. Indeed, all the algorithms for analyzing Boolean programs discussed in prior work run in time exponential in the size of the Boolean program, or worse – sometimes runtime complexity is discussed explicitly, sometimes such a discussion is omitted altogether. Moreover, no lower bounds are discussed in prior work on analyzing Boolean programs, to the best of our knowledge.

In contrast, the complexity of model checking for pushdown automata, context-free processes and recursive state machines has been studied extensively in the literature (e.g., [9,8,1,28]). However, Boolean programs can be exponentially more succinct than ordinary pushdown systems or recursive state machines. Therefore, the program complexity of model checking for Boolean programs does not follow directly from prior work on model checking for pushdown systems.

This paper aims to fill this void by providing a comprehensive study of the worst-case complexity of several basic analyses of Boolean programs, including reachability analysis, cycle detection, LTL, CTL and CTL* model checking. Furthermore, we study several natural subclasses of Boolean programs and characterize precisely the effects on the complexity of basic restrictions on the structure of the procedures or the type of the recursion: (i) deterministic vs. nondeterministic programs, (ii) hierarchical programs where there is no cycle of mutual recursion between the procedures, (iii) programs where the procedures have a bounded number of input and output arguments. In all the cases, we present algorithms (upper bounds) as well as matching lower bounds for all the problems we consider. In other words, all the algorithms presented in this paper are *optimal* in the complexity-theoretic sense.

Boolean programs correspond to recursive state machines extended with variables (ERSM for short), and can be mapped to ordinary recursive state machines (RSM) that are equivalent but exponentially larger, i.e., the use of variables, besides the syntactical convenience, allows an exponentially more succinct representation than ordinary RSM. Many times this exponential succinctness in representation results in a corresponding exponential increase in the complexity of problems. Indeed there are metatheorems in other domains (e.g., graphs represented succinctly via circuits [27]) showing that under general conditions the succinctness causes an exponential increase in complexity (for example, NP-complete problems become NEXPTIME-complete, P-complete problems become EXPTIME-complete, etc.). However, this is not the case here: the picture is much more varied and rich. As our results show, the succinctness afforded by the use of variables in the extended version of a model (recursive state machines, hierarchical state machines and their subclasses) causes in some cases an exponential jump in complexity (as one may expect), while in other cases the jump is less than exponential, and in yet other cases there is no jump at all. For example, we show that reachability analysis and LTL model checking for Boolean programs (i.e., ERSM) are EXPTIME-complete, while we know that for RSM these problems are P-complete. However, in the hierarchical case, reachability and LTL model checking for Extended Hierarchical State Machines (EHSM) are PSPACE-complete, and not EXPTIME-complete, as one might expect from the fact that for HSM (Hierarchical State Machines, without variables) these problems are still P-complete, like for RSM. Furthermore, CTL model checking for EHSM and HSM have the same complexity, it is PSPACE-complete, i.e., in this case there is no jump at all.

Similarly, there is also interesting variability in the effects that restrictions on the programs, like determinism, have on the complexity of the problems. For example, reachability analysis for deterministic Boolean programs (ERSM) is EXPTIME-complete, the same as for nondeterministic programs. However, for CTL model checking, determinism reduces the complexity by one exponential: for nondeterministic Boolean programs it is 2EXPTIME-complete, while for deterministic Boolean programs it is still EXPTIME-complete (like reachability).

As a consequence of this richness and variability in the effects of the succinctness afforded by variables and of the restrictions, one has to deal individually with the different problems, models and restrictions, and use appropriate techniques in each case to obtain the correct matching upper and lower bounds.

This paper is organized as follows. In Section 2, we formally define Boolean programs and compare them to other models of computation. In Section 3, we study the complexity of reachability analysis for Boolean programs. We also identify particular program classes for which the complexity is lower, illustrating how various features of Boolean programs contribute to the overall problem complexity. We then discuss cycle detection and LTL model checking in Section 4. In Section 5, we turn to the complexity of model checking for branching-time properties expressed in the temporal logics CTL and CTL*. Section 6 summarizes and discusses insights gained by this work. We conclude in Section 7. Proofs of theorems are given in the full paper.

2 Boolean Programs

Boolean programs are imperative programs with the usual constructs of languages like C, that have Boolean variables, and which can use nondeterminism and recursion. [5] describes in detail their syntax and defines their semantics using their control flow graphs. Boolean programs are essentially recursive state machines extended with a finite set of Boolean variables. Therefore, we will use the terms "Boolean program" and "Extended Recursive State Machine" (ERSM) interchangeably in this paper.

2.1 Syntax

Formally, a (Boolean) *Extended Recursive State Machine (ERSM)* A over a finite alphabet Σ is defined by a tuple $\langle A_1, \ldots, A_k, V \rangle$, where V is a finite set of global Boolean variables and each *procedure* A_i consists of the following pieces:

– A finite set V_i of Boolean variables that are local to the procedure A_i, a tuple $V_i^{in} \subseteq V_i$ of *input variables* and a tuple $V_i^{out} \subseteq V_i$ of *output variables*.
– A finite set N_i of *nodes* and a (disjoint) finite set B_i of *boxes*, or *call sites*.

- A labeling $Y_i : B_i \to \{1, \ldots, k\}$ that assigns to every box an index of one of the procedures (component machines), $A_1, \ldots, A_k$, and a pair of mappings $\beta_i^{in}, \beta_i^{out}$ which assign to each box $b \in B_i$ two tuples $\beta_i^{in}(b), \beta_i^{out}(b)$ of variables in V_i that are respectively the input and output arguments of the recursive call represented by the box b, where $|\beta_i^{in}(b)| = |V_{Y_i(b)}^{in}|$ and $|\beta_i^{out}(b)| = |V_{Y_i(b)}^{out}|$.
- A set of *entry* nodes $En_i \subseteq N_i$, and a set of *exit* nodes $Ex_i \subseteq N_i$.
- A *transition relation* δ_i, where transitions are of the form (u, G, σ, C, v) where (1) the source u is either a node of $N_i \setminus Ex_i$, or a pair (b, x), where b is a box in B_i and x is an exit node in Ex_j for $j = Y_i(b)$; (2) the guard G is a Boolean predicate on the variables in $V_i \cup V$; (3) the label σ is in Σ; (4) the command C assigns new Boolean values to the variables in $V_i \cup V$ as a function of the old values; and (5) the destination v is either a node in N_i or a pair (b, e), where b is a box in B_i and e is an entry node in En_j for $j = Y_i(b)$.

We will use the term *ports* to refer to pairs $(b, e), (b, x)$ consisting of a box b of a procedure A_i and corresponding entry nodes e and exit nodes x of the procedure A_j called by b. We will use the term *vertices* of A_i to refer to its nodes and the ports of its boxes that participate in some transition. We will often refer to a vertex (b, e) as a *call vertex* and (b, x) as a *return vertex*.

We define the *size* $|A|$ of an ERSM A to be the sum of the total numbers of nodes, boxes, transitions and variables of A.

Remarks: 1. In the above definition we have allowed procedures to have multiple entries (initial nodes) and exits (final nodes). In the presence of variables, this is strictly speaking not necessary, i.e., ERSMs where every procedure has a single entry and exit are equally expressive, because we can use extra input and output variables to specify different entries and exits. In fact, in a straightforward translation of the code of a Boolean program to an ERSM, the procedures will have a single entry and exit. A statement like $y := A_j(x)$ in a procedure A_i corresponds to a box b with $Y_i(b) = j$, $\beta_i^{in}(b) = x$, and $\beta_i^{out}(b) = y$. We have allowed multiple entries and exits here for consistency with the definition of standard RSMs that do not have variables [1], where the multiplicity of entries and exits is essential.
2. It is convenient syntactically for procedures to receive inputs and return outputs, although in the presence of global variables it is not really essential to have explicitly input and output variables: a value passed as argument to a procedure can be modeled using a global variable which is assigned the argument value just before the procedure call and then copied immediately after the start of the called procedure into a local variable of that procedure. Similarly, a return value of a procedure can be modeled with a global variable which is assigned the return value just before the return and then copied immediately after the return into the local state of the calling procedure.
3. The syntax of the guards and commands of the transitions in the definition is left flexible. For the complexity upper bounds, we assume that the guards and commands are arbitrary predicates and functions respectively that can be evaluated in polynomial time. For the lower bound constructions, the guards are simple equality conditions, and the commands are simple assignments.

4. In the above definition, all variables are Boolean. More generally, we could define ERSMs whose variables have other domains. If all the variables have finite domains, we can clearly encode them with Boolean variables, and the results of the paper apply.

In what follows, we will represent ERSMs using pseudo-code.

2.2 Semantics

To define the executions of ERSMs, we first define the global states and transitions associated with an ERSM. Let $\overline{X}$ denote a mapping that associates a value to each variable in a set X of variables. We assume all Boolean variables have a unique default initial value. A (global) *state* of an ERSM $A = \langle A_1, \ldots A_k, V\rangle$ is a tuple $\langle(b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V})\rangle$ where $b_1, \ldots, b_r$ are boxes, $\overline{V_1}, \ldots, \overline{V_r}, \overline{V_{r+1}}$ are value assignments to local variables, u is a node, and $\overline{V}$ assigns a value to every global variable. Equivalently, a state can be viewed as a string, and the set Q of global states of A is $(B \times \overline{V'})^*(N \times \overline{V'} \times \overline{V})$, where $B = \cup_i B_i$, $V' = \cup_i V_i$ and $N = \cup_i N_i$. Consider a state $\langle(b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V})\rangle$ such that $b_i \in B_{j_i}$ for $1 \leq i \leq r$ and $u \in N_j$. Such a state is *well-formed* if $Y_{j_i}(b_i) = j_{i+1}$ and $V_i = V_{j_i}$ for $1 \leq i < r$, and if $Y_{j_r}(b_r) = j$ and $V_{r+1} = V_j$. A well-formed state of this form corresponds to the case when the control is inside the component A_j, which was entered via box b_r of component A_{j_r} (the box b_{r-1} gives the context in which A_{j_r} was entered, and so on). Henceforth, we assume states to be well-formed. Given a state $\langle(b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V})\rangle$, we will sometimes refer to $\langle(b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r})\rangle$ as the *call stack*, or stack, in that state.

We assume a call-by-value model for the procedure calls. We define a (global) transition relation δ among the global states of A as follows. Let $s = \langle(b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u, \overline{V_{r+1}}, \overline{V})\rangle$ be a state with $u \in N_j$ and $b_r \in B_m$. Then, $(s, \sigma, s') \in \delta$ iff one of the following holds:

1. $(u, G, \sigma, C, u') \in \delta_j$ for a node u' of A_j, $G(\overline{V_{r+1}}, \overline{V})$ evaluates to true, $C(\overline{V_{r+1}}, \overline{V}) = (\overline{V_{r+1}}', \overline{V}')$, and $s' = \langle(b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (u', \overline{V_{r+1}}', \overline{V}')\rangle$. This case is when the control stays within the component A_j.
2. $(u, G, \sigma, C, (b', e)) \in \delta_j$ for a box b' of A_j, $G(\overline{V_{r+1}}, \overline{V})$ evaluates to true, $C(\overline{V_{r+1}}, \overline{V}) = (\overline{V_{r+1}}', \overline{V}')$, and $s' = \langle(b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (b', \overline{V_{r+1}}'), (e, \overline{V_{r+2}}', \overline{V}')\rangle$, where $\overline{V_{r+2}}'$ denotes an initial value assignment for the local variables in $V_{Y_j(b')}$ of the procedure corresponding to box b', in which the input variables $V^{in}_{Y_j(b')}$ have value equal to the value of the variables $\beta^{in}_j(b')$ in $\overline{V_{r+1}}'$. This case is when a new component is entered via a box of A_j.
3. u is an exit-node of A_j, $((b_r, u), G, \sigma, C, u') \in \delta_m$ for a node u' of A_m, $\hat{V_r}$ is the assignment to the local variables of A_m in which the variables of $\beta^{out}_m(b_r)$ have value equal to that of the output variables V^{out}_j of A_j in $\overline{V_{r+1}}$ and the rest of the variables have the same value as in $\overline{V_r}$, $G(\hat{V_r}, \overline{V})$ evaluates to true, $C(\hat{V_r}, \overline{V}) = (\overline{V_r}', \overline{V}')$, and $s' = \langle(b_1, \overline{V_1}), \ldots, (b_{r-1}, \overline{V_{r-1}}), (u', \overline{V_r}', \overline{V}')\rangle$. This case is when the control exits A_j and returns back to A_m.

4. u is an exit-node of A_j, $((b_r, u), G, \sigma, C, (b', e)) \in \delta_m$ for a box b' of A_m, $\hat{V_r}$ is the assignment to the local variables of A_m in which the variables of $\beta_m^{out}(b_r)$ have value equal to that of the output variables V_j^{out} of A_j in $\overline{V_{r+1}}$ and the rest of the variables have the same value as in $\overline{V_r}$, $G(\hat{V_r}, \overline{V})$ evaluates to true, $C(\hat{V_r}, \overline{V}) = (\overline{V_r}', \overline{V}')$, and $s' = \langle (b_1, \overline{V_1}), \ldots, (b_{r-1,} \overline{V_{r-1}}), (b', \overline{V_r}'), (e, \overline{V_{r+1}}', \overline{V}') \rangle$, where $\overline{V_{r+1}}'$ denotes an initial value assignment for the local variables in $V_{Y_m(b')}$ of the procedure corresponding to box b', in which the input variables $V_{Y_m(b')}^{in}$ have value equal to the value of the variables $\beta_m^{in}(b')$ in $\overline{V_r}'$. This case is when the control exits A_j and enters a new component via a box of A_m.

The *Labeled Transition System (LTS)* $T_A = (Q, \Sigma, \delta)$ is called the *"unfolding" of* A. The set Q of reachable states can be infinite. For a state s of the LTS T_A and a node v of A, $s \Rightarrow v$ denotes that s can reach some state $\langle (b_1, \overline{V_1}), \ldots, (b_r, \overline{V_r}), (v, \overline{V_{r+1}}, \overline{V}) \rangle$ in T_A whose node is v.

2.3 Special Classes

ERSMs generalize several other well-known models of computation.

- A *Recursive State Machine (RSM)* is an ERSM with no Boolean variables, i.e., where V and the sets V_i are all empty, the guards G are all vacuously $true$, and the commands C do not modify the value of any variable.
- An *Extended Hierarchical State Machine (EHSM)* is an ERSM with no cycle of recursive calls between the procedures, i.e., where every procedure A_i can only call a procedure A_j with $j > i$, i.e., we have $\forall i : \forall b \in B_i : Y_i(b) > i$.
- A *Hierarchical State Machine (HSM)* is an EHSM with no Boolean variables.
- An *Extended Finite State Machine (EFSM)* is an ERSM (or EHSM) with a single procedure A_1 and no boxes.
- A *Finite State Machine (FSM)* is an EFSM with no Boolean variables.

A procedure or machine A_i is called *single-entry* when it has a single entry node e, i.e., when $En_i = \{e\}$. Similarly, a procedure or machine A_i is called *single-exit* when it has a single exit node x, i.e., when $Ex_i = \{x\}$. An ERSM is single-entry or single-exit if all its procedures are. As mentioned earlier, any ERSM can be transformed to an equivalent single-entry, single-exit ERSM by introducing additional variables. This is not the case for RSMs.

A Boolean program A is called *input/output bounded*, or *I/O bounded* for short, if the number of the input and output variables of every procedure, and the number of global variables are $O(\log |A|)$ (i.e., upper bounded by $c \cdot \log |A|$ for some fixed constant c). The procedures themselves can be arbitrarily large and complex, and use an arbitrary number of local variables. The I/O bounded property characterizes programs where there is a limited amount of information communicated between the different procedures.

A procedure A_i is called *acyclic* if the graph $(N_i \cup B_i, E_i)$ is acyclic, where E_i contains an edge from a node u or box b to another node u' or box b' iff δ_i contains a transition from u or a vertex of b to u' or a vertex of b' (regardless of the guard and command of the transition). An ERSM is acyclic iff all its procedures are.

A procedure is called *deterministic* if, for all its vertices, the guards of all its transitions at that vertex are mutually exclusive. In that case, each state of that procedure can have at most one successor state. A program is deterministic if all its procedures are deterministic. Usual programs (without abstraction) are deterministic.

2.4 Expansion of an ERSM

Given an ERSM $A = \langle A_1, \dots A_k, V\rangle$, we can construct an RSM $A' = \langle A'_1, \dots A'_k\rangle$ (without variables) that is equivalent to A, in the sense that their unfoldings T_A and $T_{A'}$ are identical. The construction of A' involves combining every vertex of each procedure of A with every valuation for the global and local variables (see the full paper for the detailed construction). The RSM A' is in general exponentially larger than A. In particular, if $m = \max_i |V \cup V_i|$ then the size $|A'|$ of the RSM A' is (at most) $|A| \cdot 2^m$. We call A' the *expanded* RSM corresponding to A.

3 Reachability

Let $Init$ denote a given set of initial states, consisting of some entry nodes together with specified valuations for the variables in the scope of their procedures. Given an ERSM $A = \langle A_1, \dots A_k, V\rangle$ and such a set $Init$, let $Init \Rightarrow v$ denote that for some $s \in Init$, $s \Rightarrow v$. Our goal in simple reachability analysis is to determine whether a specific target node t is in the set $\{v \mid Init \Rightarrow v\}$ of reachable vertices. In this section, we study the complexity of the reachability analysis problem for ERSMs and several special cases.

Theorem 1. *Reachability analysis for ERSMs is EXPTIME-complete. Furthermore, this holds even for deterministic, acyclic ERSMs.*

Sketch: Membership in EXPTIME follows essentially from previous work (e.g., [4,1]). Given a ERSM A, we can construct the corresponding expanded RSM A', which has size (at most) exponential in A. Since reachability analysis for RSMs can be solved in polynomial time (cubic in the general case, and linear for single-entry or single-exit RSMs to be precise [1]), we obtain an algorithm with EXPTIME complexity overall.

For the hardness part, we reduce the acceptance problem for 1-tape alternating polynomial space machines, which is known to be EXPTIME-complete [10], to reachability analysis of ERSMs. Figure 1 shows a Boolean program (left) simulating an alternating PSPACE machine. The proof is given in the full paper. ■

The Boolean program of Figure 1 is deterministic and acyclic, so these features do not make a dramatic difference in the complexity of ERSM reachability analysis. Note that the procedure Acc in the program of Figure 1 is recursive and passes a linear amount of information in each call. We now show that restricting the use of recursion or the amount of I/O information reduces the complexity to a lower class.

In the hierarchical case, reachability analysis becomes PSPACE-complete, thus, no worse than simple EFSMs. Note that if we expand the EHSM to an (exponentially larger) HSM and apply the HSM reachability algorithm, the resulting algorithm will have exponential space complexity, and this is probably inherent in that approach since reachability for HSM is P-complete [3].

```
procedure Top()
{
  if Acc($q_0$, 0, Initial Tape)
     then print(``M accepts'');
}

bool Acc(state q, head location h, Tape T)
{
  if (q in $Q_T$) then return true;
  if (q in $Q_F$) then return false;

  bool res;
  if (q in $Q_\exists$) then res = false;
  else res = true; // case (q in $Q_\forall$)

  for each (q',s,D) in $\delta_M$(q,T[h])
  {
    compute new tape location h' and tape T';
    if (q in $Q_\exists$) then res=res$\vee$Acc(q',h',T');
    else res=res$\wedge$Acc(q',h',T');
  }
  return res;
}
```

```
procedure Top()
{
  if SAT[0]()
     then print(``$\psi$ is SAT'');
}

bool SAT[n](bool $x_1,\ldots,x_n$)
{
  return ($\phi(x_1,\ldots,x_n)$); // evaluate $\phi$
}

// if i is odd, $x_{i+1}$ is after $\forall$ in $\psi$
bool SAT[i](bool $x_1,\ldots,x_i$)
{
  return(SAT[i+1]($x_1,\ldots,x_i$,0)
         $\wedge$ SAT[i+1]($x_1,\ldots,x_i$,1));
}

// if i is even, $x_{i+1}$ is after $\exists$ in $\psi$
bool SAT[i](bool $x_1,\ldots,x_i$)
{
  return(SAT[i+1]($x_1,\ldots,x_i$,0)
         $\vee$ SAT[i+1]($x_1,\ldots,x_i$,1));
}
```

Fig. 1. Boolean programs simulating an alternating PSPACE machine M (left) and for checking satisfiability of the QBF formula $\psi = \exists x_1 \forall x_2 \exists x_3 \ldots Q x_n \phi(x_1,\ldots,x_n)$ (right)

Theorem 2. *Reachability analysis for EHSMs is PSPACE-complete. Furthermore, the problem remains PSPACE-complete for deterministic, acyclic EHSMs.*

Sketch: Membership in PSPACE follows from nondeterministically simulating a computation that reaches the target node using polynomial space, and applying Savitch's theorem to make it deterministic. Since reachability analysis is already known to be PSPACE-hard for EFSMs, PSPACE-hardness for the more general EHSMs follows immediately. Moreover, the problem remains PSPACE-complete for EHSMs that are deterministic and acyclic. For this purpose, we reduce Quantified Boolean Formula (QBF) satisfiability (QSAT), known to be PSPACE-complete, to EHSM reachability: Figure 1 shows a deterministic acyclic hierarchical Boolean program (on the right) for checking the satisfiability of a QBF formula ψ of the form $\exists x_1 \forall x_2 \exists x_3 \ldots Q x_n \phi(x_1,\ldots,x_n)$. The proof is given in the full paper. ■

For acyclic EFSM and, more generally, for acyclic EHSMs where the depth of the hierarchy is bounded by a constant, the complexity of reachability analysis is reduced further to NP-complete.

Theorem 3. *Reachability analysis for acyclic EHSMs of bounded depth is NP-complete.*

We now consider the subclass of I/O bounded Boolean programs, and show that the complexity is lower.

Theorem 4. *Reachability analysis for I/O bounded deterministic acyclic EHSMs is in P.*

Theorem 5. *Reachability analysis for I/O bounded* nondeterministic *acyclic EHSMs is NP-complete.*

Class of Program	Restriction	General Case	I/O Bounded
ERSM		EXPTIME	PSPACE
EHSM		PSPACE	PSPACE
EHSM	nondeterministic acyclic	PSPACE	NP
EHSM	deterministic acyclic	PSPACE	P

Fig. 2. Complexity of reachability analysis

Theorem 6. *Reachability analysis for I/O bounded* cyclic *EHSMs is PSPACE-complete.*

Moreover, in the world of I/O bounded programs, reachability analysis for ERSMs is not more expensive than for EHSMs or just EFSMs.

Theorem 7. *Reachability analysis for I/O bounded ERSMs is PSPACE-complete.*

Sketch: The algorithm involves doing first a partial expansion of the ERSM where we only expand in each procedure the input and output variables and the global variables. Then we remove the boxes from the procedures, yielding a collection of EFSMs, and solve iteratively a sequence of EFSM reachability problems to infer incrementally the reachabilities between the expanded entries and exits of the procedures. Finaly we construct a final single EFSM $\hat{C}$ that incorporates the entry-exit reachabilities and interconnects the procedures, and solve an EFSM reachability problem on $\hat{C}$ to compute all the vertices that are reachable from the initial set $Init$. See the full paper for details. ■

Most of the results of this section are summarized in Figure 2.

4 LTL Model Checking

We now consider linear time properties expressed in Linear Temporal Logic (LTL) or using Büchi automata. Formulas of LTL are built from a finite set $Prop$ of atomic propositions using the usual Boolean operators $\neg$, $\vee$, $\wedge$, the unary temporal operators X (next), and the binary operator U (until). A Büchi automaton is a finite (nondeterministic) automaton on infinite words that accepts a word w iff it has a run on w that visits the subset of accepting states infinitely often. Every LTL formula ϕ can be translated to an equivalent Büchi automaton D_ϕ over the alphabet $\Sigma = 2^{Prop}$ (the translation may increase exponentially the size in general). The LTL or automaton model checking problem is to determine whether all computations of a given Kripke structure T (starting from designated initial states) satisfy a given LTL formula ϕ or are accepted by a Büchi automaton D. We refer to [13] for detailed background on LTL, automata and model checking. In our case the Kripke structure is the unfolding T_A of a given ERSM A over $\Sigma = 2^{Prop}$.

All the results for reachability of the last section extend to model checking of all linear time properties, with the same dependency of the complexity on the size of the program (this is called the *program complexity*) in all the cases, i.e., for general ERSMs as well as for their subclasses. The dependence of the complexity on the size of the specification is polynomial for automata specifications and exponential for LTL (as is the case for model checking of even nonrecursive finite state structures). Rather than list the individual results, we state them collectively in the following:

Theorem 8. *The program complexity of model checking linear time properties of ERSMs is the same as that given for reachability analysis in the last section, for all the considered classes of ERSMs.*

Due to space constraints we will not give the proofs for the various classes. Roughly speaking, LTL model checking involves forming the product ERSM $\hat{A}$ of the ERSM with an automaton $D_{\neg\phi}$ representing the negation of the property, and testing whether (the unfolding of) $\hat{A}$ has a reachable cycle that contains an accepting state or has an accepting computation path where the stack grows without bound. Both of these cases can be solved using suitable reachability problems. The specifics of the algorithms depend on the class of ERSMs; in some cases this is easy, while in others it is nontrivial.

5 Branching-Time Properties

We now consider the verification of properties expressed in the branching-time logic CTL [12]. CTL allows quantification over computations of a system, such as "along some computation, eventually p" or "along all computations, eventually p". The temporal logic CTL uses the temporal operators U (until), X (nexttime) and the existential path quantifier E, in addition to the operators $\neg$ (not) and $\vee$ (or). We use the standard abbreviations Ap (for all paths p) for $\neg E\neg p$, Fp (eventually p) for $trueUp$, and Gp (always p) for $\neg F\neg p$. See [13] for a detailed description of the syntax and semantics of CTL.

The *CTL model checking problem* is to decide whether a Kripke structure satisfies a CTL formula [12]. In our context, unfoldings of ERSMs will be used as Kripke structures.

Theorem 9. *The program complexity of CTL model checking for ERSMs is 2EXPTIME-complete.*

Sketch: Given an ERSM A, we can build an exponentially larger RSM A' such that their unfoldings T_A and $T_{A'}$ are identical. Then, we can use the CTL model checking algorithm for RSMs discussed in [1], whose running time can be exponential in the size of the RSMs. Overall, we thus obtain an algorithm with 2EXPTIME complexity.

To prove 2EXPTIME-hardness, we reduce the acceptance problem for 1-tape alternating exponential space machines, which is known to be 2EXPTIME-complete [10], to CTL model checking of ERSMs. Given an alternating EXPSPACE machine M and an input x, we construct a Boolen program P that simulates the computations of M on x. A problem here is that the exponentially large tape cannot be passed as an argument (unlike the proof of Theorem 1). The main idea to address this is to have the program nondeterministically guess continuously the contents of the tape, cell by cell, and store it in the stack. Another part of the program may nondeterministically at any point stop the computation and backtrack to try to check whether the content of a particular cell is consistent with the previous configuration. The constructed CTL formula φ is a fixed formula that says that there is a computation of the program P that leads to acceptance and if we were to do any check along the way it would turn out ok. We show that the EXPSPACE alternating machine M accepts an input x if and only if P satisfies φ; see the full paper for the details of the construction and the proof. ■

The 2EXPTIME-hardness proof relies on the Boolean program to be nondeterministic. Indeed, we now prove that CTL model checking for *deterministic* Boolean programs is "only" EXPTIME-complete.

Theorem 10. *The program complexity of CTL model checking for* deterministic *ERSMs is EXPTIME-complete.*

The proof involves the development of a new efficient algorithm for CTL model checking of deterministic RSM showing the following:

Theorem 11. *CTL model checking for* deterministic *multi-exit RSMs can be done in time linear in the size of the structure.*

Sketch: Given a deterministic multi-exit RSM A we show how to construct in linear time an equivalent single-exit RSM A", and then we use the linear-time algorithm for CTL model checking of single-exit RSMs from [1]. The construction of A" involves two phases. In the first phase, we compute for each initial node incrementally all the reachable vertices, and for each reachable vertex, we compute whether it can reach an exit node of its component and which one. This has to be done carefully to ensure that nonterminating computations are cut off promptly and that every reachabe vertex is processed only at most twice, and thereby achieve linear time in the number of reachable vertices. In the second phase, we construct in linear time from the information of Phase 1 a single-exit RSM A" that contains several procedures for each component of A with the property that every reachable vertex and edge of A appears in exactly one procedure of A", and A" has no other vertices and edges. Furthermore, the reachable parts of the unfoldings of A and A" are identical. See the full paper for the details. ■

Theorem 10 can be shown then by expanding the given deterministic ERSM A to a RSM and applying the algorithm of Theorem 11. The expansion can be done in fact on the fly, only to the extent that is needed, starting from the set $Init$ of initial states, so that the whole CTL model checking algorithm takes time proportional to the number of reachable vertices in the expanded RSM.

The algorithm used in the proof of Theorem 10 is useful also to reduce the complexity of reachability and LTL model checking for deterministic ERSM, from cubic to linear in the number of reachable expanded vertices. (Of course we cannot expect an exponential reduction in view of Theorem 1.)

Obviously, Theorem 11 implies that CTL model checking of deterministic multi-exit HSMs can also be done in linear time (since HSMs are special RSMs), in contrast with the general case of nondeterministic multi-exit HSMs for which the program complexity of CTL model checking is known to be PSPACE-complete [3].

In the case of EHSMs, we can show that determinism does not help reduce the program complexity of CTL model checking compared to the nondeterministic case. However, and perhaps surprisingly, the program complexity of CTL model checking for EHSMs is the same as for HSMs: it is also PSPACE-complete.

Theorem 12. *The program complexity of CTL model checking for EHSMs is PSPACE-complete.*

Since EFSMs are special cases of EHSMs, the previous PSPACE upper bound carries over to EFSMs, and we have the following.

Class of Program	Restriction	LTL	CTL
FSM		Linear	Linear
EFSM		PSPACE	**PSPACE**
HSM		Linear	PSPACE
HSM	deterministic	Linear	**Linear**
EHSM		**PSPACE**	**PSPACE**
EHSM	deterministic	**PSPACE**	**PSPACE**
RSM		Cubic	EXPTIME
RSM	deterministic	**Linear**	**Linear**
ERSM		**EXPTIME**	**2-EXPTIME**
ERSM	deterministic	**EXPTIME**	**EXPTIME**

Fig. 3. Complexity bounds in the size of the program. The new bounds from this paper are highlighted in bold.

Corollary 1. *The program complexity of CTL model checking for EFSMs is PSPACE-complete.*

Since EFSMs are standard, the last result might be already known, but we do not know if it is stated somewhere in the literature.

Finally we note that all the algorithms of this section apply also to the more powerful branching time logic CTL* (see [13] for a definition) with exactly the same complexity:

Theorem 13. *The program complexity of CTL* model checking is as follows:*
1. For ERSMs it is 2EXPTIME-complete.
2. For deterministic ERSMs it is EXPTIME-complete.
3. For EHSMs it is PSPACE-complete.

6 Discussion

6.1 Summary of Results

Figure 2 summarizes the results for reachability and linear time properties. For general Boolean programs (ERSMs) the problems are EXPTIME-complete which means that the analysis provably requires exponential time in the worst-case. Since even reachability of simple EFSMs (which have no recursion) is PSPACE-complete, we cannot hope for better than PSPACE for programs with variables that include EFSMs. As we see, PSPACE suffices for important subclasses including EHSM (hierarchical recursion) and I/O bounded ERSM (bounded communication). For the I/O bounded class, the complexity is reduced further in more restricted cases.

Figure 3 summarizes the results regarding the program complexity of LTL and CTL (and CTL*) model checking for general (nondeterministic) and deterministic ERSMs and EHSMs and their counterparts RSM, HSM that have no variables. New results from this work are highlighted in bold.

From Figure 3, we observe that the program complexity of CTL model checking for deterministic programs is exponentially better than for nondeterministic ones, *except* for EHSMs where the complexity does not change. In practice, this means that whenever it is possible to *hoist* nondeterministic choices in a Boolean programs to its initial

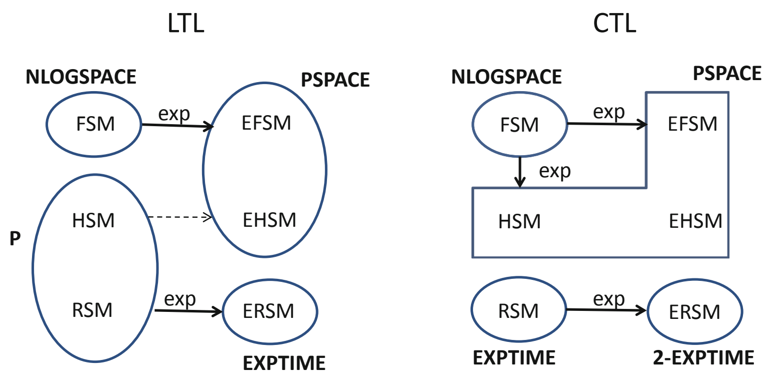


Fig. 4. Visual summary for the program complexity of LTL and CTL model checking

states, then the program effectively becomes deterministic and CTL model checking can be exponentially faster in the size of the program. On the other hand, for LTL model checking, determinism decreases the complexity more modestly, by a polynomial amount.

Figure 4 compares the program complexity of LTL and CTL model checking for the main (no restriction) classes of programs considered in Figure 3. From this figure, we make the following observations.

- Adding Boolean variables (extension "E") to programs increases the program complexity of model checking *except* for HSMs and CTL model checking.
- Adding hierarchy to EFSMs does not increase the program complexity of model checking for LTL or CTL. Adding further full recursion increases somewhat the complexity for LTL, but much more drastically (more than exponentially) for CTL.
- For a fixed program class, CTL model checking can be exponentially more expensive in the size of the program than LTL model checking, *except* in the case of EFSMs and EHSMs (where the complexity remains PSPACE-complete) and in the FSM case (where the complexity is linear in both cases).

6.2 Comparison with Pushdown Model Checking

In [1], it is shown that every *RSM* is bisimilar to a *pushdown system* (also called pushdown automaton). Therefore, the program complexity of model checking for RSMs and pushdown systems is the same. Since Boolean programs can be exponentially more succinct than ordinary pushdown systems or recursive state machines, the program complexity of model checking for Boolean programs does not follow directly from prior work on model checking for traditional pushdown systems. The same comment applies to prior work on hierarchical systems (e.g. [3,2,20,26]).

[17] defines "symbolic pushdown systems", which are pushdown systems extended with variables in the control states and the stack symbols, it shows how to derive such a system from a Boolean program, and gives an algorithm for LTL model checking (the algorithm has exponential complexity). No lower bound is given on the complexity of the problem.

6.3 Impact on Logic Encodings

The complexity results presented in our work also shed new light on how to represent classes of Boolean programs using logic, and the abilities and limitations of different logics in this respect.

An approach to symbolic program analysis consists in representing the program by a logic formula, possibly generated incrementally, and then reducing reachability analysis and property checking to a satisfiability or validity check for the corresponding logic performed using a SAT or SMT solver. This is the methodology used in *verification-condition generation* [16,18,6] and *SAT/SMT-based bounded model checking* [11,14].

For a *polynomial-size* logic encoding of a specific class of programs, it is necessary to use a sufficiently-expressive logic. For instance, consider the EHSM case. Theorem 2 states that reachability analysis for EHSMs is PSPACE-complete. This suggests that a polynomial-size encoding for EHSMs is possible using a logic like QBF since satisfiability for QBF is also PSPACE-complete. (Such an encoding is indeed possible.) This also proves that a polynomial-size encoding in a less expressive logic, such as propositional logic, is impossible: a (precise) translation from EHSMs to propositional logic may result in formulas that are exponentially larger than the program. In contrast, Theorems 3 and 5 identify specific classes of EHSMs for which reachability analysis is "only" NP-complete and for which precise polynomial-size encodings to propositional logic are possible (as satisfiability for propositional logic is NP-complete).

7 Conclusion

Boolean programs are a simple, natural and popular abstract domain for static-analysis-based software model checking. This paper presents the first comprehensive study of the worst-case complexity of several basic analyses of Boolean programs, including reachability analysis, cycle detection, and model checking for the temporal logics LTL, CTL and CTL*. We also studied several natural classes of Boolean programs which are easier to analyze. We presented matching lower and upper bounds for all these problems. The overall picture is quite rich and varied and required a range of different techniques. The results help explain what features of Boolean programs contribute to the overall worst-case complexity. For instance, nondeterminism does not impact drastically the complexity of reachability analysis for Boolean programs (it increases it only polynomially) while it impacts much more significantly (exponentially) the program complexity of CTL model checking.

Acknowledgments. Research partially supported by NSF Grant CCF-1017955.

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of Recursive State Machines. ACM Trans. on Programming Languages and Systems (TOPLAS) 27(4), 786–818 (2005)
2. Alur, R., Kannan, S., Yannakakis, M.: Communicating Hierarchical State Machines. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 169–178. Springer, Heidelberg (1999)

3. Alur, R., Yannakakis, M.: Model Checking of Hierarchical State Machines. ACM TOPLAS 23(3), 273–303 (2001)
4. 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)
5. Ball, T., Rajamani, S.K.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
6. Barnett, M., Leino, K.R.M.: Weakest Precondition of Unstructured Programs. In: Proc. PASTE (Program Analysis For Software Tools and Engineering), pp. 82–87 (2005)
7. Basler, G., Kroening, D., Weissenbacher, G.: SAT-Based Summarization for Boolean Programs. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 131–148. Springer, Heidelberg (2007)
8. 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)
9. Burkart, O., Steffen, B.: Model Checking for Context-Free Processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)
10. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the ACM 28(1), 114–133 (1981)
11. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design 19(1), 7–34 (2001)
12. 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)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
14. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral Consistency of C and Verilog Programs using Bounded Model Checking. In: Design Automation Conference (DAC), pp. 368–371. ACM (2003)
15. Cook, B., Podelski, A., Rybalchenko, A.: Termination Proofs for Systems Code. In: Proceedings of PLDI 2006, pp. 415–426 (2006)
16. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Comm. of the ACM 18, 453–457 (1975)
17. Esparza, J., Schwoon, S.: A BDD-Based Model Checker for Recursive Programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
18. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings of PLDI 2002, pp. 234–245 (2002)
19. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In: Proc. POPL, pp. 43–55 (2010)
20. Goller, S., Lohrey, M.: Fixpoint Logics over Hierarchical Structures. Theory Comp. Sys. 48(1), 93–131 (2011)
21. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
22. Gurfinkel, A., Wei, O., Chechik, M.: YASM: A Software Model-Checker for Verification and Refutation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 170–174. Springer, Heidelberg (2006)
23. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proceedings of POPL 2002, Portland, pp. 58–70 (January 2002)

24. Kupferman, O., Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Branching-Time Model Checking. Journal of the ACM 47(2), 312–360 (2000)
25. Leino, K.R.M.: A SAT Characterization of Boolean-Program Correctness. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 104–120. Springer, Heidelberg (2003)
26. Lohrey, M.: Model-checking hierarchical structures. J. Comp. Sys. Sc. 78(2), 461–490 (2012)
27. Papadimitriou, C.H., Yannakakis, M.: A Note on Succinct Representation of Graphs. Inf. and Comp. 71(3), 181–185 (1986)
28. Walukiewicz, I.: Model Checking CTL Properties of Pushdown Systems. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 127–138. Springer, Heidelberg (2000)

Weighted Pushdown Systems with Indexed Weight Domains

Yasuhiko Minamide

Faculty of Engineering, Information and Systems
University of Tsukuba

Abstract. The reachability analysis of weighted pushdown systems is a very powerful technique in verification and analysis of recursive programs. Each transition rule of a weighted pushdown system is associated with an element of a bounded semiring representing the weight of the rule. However, we have realized that the restriction of the boundedness is too strict and the formulation of weighted pushdown systems is not general enough for some applications.

To generalize weighted pushdown systems, we first introduce the notion of stack signatures that summarize the effect of a computation of a pushdown system and formulate pushdown systems as automata over the monoid of stack signatures. We then generalize weighted pushdown systems by introducing semirings indexed by the monoid and weaken the boundedness to local boundedness.

1 Introduction

The reachability analysis of weighted pushdown systems is a very powerful technique in verification and analysis of recursive programs [RSJM05]. Each transition rule of a weighted pushdown system is associated with an element of a semiring representing the weight of the rule. To guarantee termination of the analysis, the semiring of the weight must be bounded: there should be no infinite descending sequence of weight. However, recently, we have realized that this restriction of the boundedness is too strict and the formulation of weighted pushdown systems is not general enough for some applications. For the two applications below, the standard algorithm for the reachability analysis of weighted pushdown systems actually works and terminates. However, they require semirings that are not bounded and thus the standard framework of weighted pushdown systems cannot guarantee termination.

The first application is the reachability analysis of conditional pushdown systems. Conditional pushdown systems extend pushdown systems with the ability to check the whole stack content against a regular language [EKS03, LO10]. We proposed an algorithm of their reachability analysis in our previous work on the analysis of HTML 5 parser specification [MM12]. After the development of the algorithm, we realized that the algorithm can be considered as the reachability analysis of weighted pushdown systems. However, it required an unbounded semiring.

The second application is the analysis of recursive programs with local variables. For the efficient analysis of recursive programs, Suwimonteerabuth proposed an encoding of local variables into weight implemented with BDDs [Suw09]. The weight has a structure depending on a configuration of stack and requires a semiring that is not bounded.

To generalize weighted pushdown systems, we first introduce *stack signatures* that summarize the effect of a computation of a pushdown system as a pair of words over stack alphabet. A stack signature w_1/w_2 represents a computation of a pushdown system that popes w_1 and pushes w_2 as its total effect. We show that the set of stack signatures forms an ordered monoid, *i.e.*, a monoid that is equipped with a partial order compatible with the multiplication of the monoid. We then formulate pushdown systems as automata over the monoid of stack signatures.

We extend the structure of weight by introducing semirings indexed by a monoid element. Weighted pushdown systems are generalized to those over a semiring indexed by the monoid of stack signatures. We show that the reachability analysis of weighted pushdown systems can be refined to those over an indexed semiring and the boundedness can be replaced with the *local boundedness*.

Finally, we show two applications of weighted pushdown systems over a semiring indexed by stack signatures. The first one is a simplified version of the structure used by Suwimonteerabuth to encode local variables of a recursive program. The other is an indexed semiring to encode the reachability analysis of conditional pushdown systems into that of weighted pushdown systems. Since both of these indexed semirings are locally bounded, our framework guarantees termination of the two analyses.

2 Semirings and Weighted Automata

We first review the definitions of semirings and weighted automata.

Definition 1. *A semiring is a structure* $\mathcal{S} = \langle D\,, \oplus\,, \otimes\,, 0\,, 1\rangle$ *where* D *is a set,* 0 *and* 1 *are elements of* D*,* $\oplus$ *and* $\otimes$ *are binary operations on* D *such that*

1. $\langle D, \oplus, 0\rangle$ *is a commutative monoid.*
2. $\langle D, \otimes, 1\rangle$ *is a monoid.*
3. $\otimes$ *distributes over* $\oplus$*.*

$$(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z) \qquad x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$$

4. 0 *is an annihilator with respect to* $\otimes$*:* $0 \otimes x = 0 = x \otimes 0$ *for all* $x \in D$*.*

We say that a semiring $\mathcal{S}$ is *idempotent* if its addition $\oplus$ is idempotent (i.e., $a \oplus a = a$). For an idempotent semiring $\langle D\,, \oplus\,, \otimes\,, 0\,, 1\rangle$, $\langle D, \oplus\rangle$ can be considered as a join semilattice[1]. Then, the partial order $\sqsubseteq$ is defined by $a \sqsubseteq b$ iff $a \oplus b = b$

[1] In [RSJM05], it is considered as a meet semilattice.

for an idempotent semiring. We say that an idempotent semiring is *bounded* if there are no infinite ascending chains with respect to $\sqsubseteq$.

In this paper, we consider weighted automata without initial and final states.

Definition 2. *A weighted automaton $\mathcal{A}$ over an idempotent semiring $\mathcal{S}$ and an alphabet Γ is a structure $\langle \Gamma, Q, E\rangle$ where Q is a finite set of states, $E : Q \times \Gamma \times Q \to \mathcal{S}$ is a set of transition rules each of which associates an element in $\mathcal{S}$ as weight.*

For weighted automata over alphabet Γ and semiring $\mathcal{S} = \langle D, \oplus, \otimes, 0, 1\rangle$, we introduce the transition relation of the form $q \xrightarrow{w\,|\,a} q'$ where $w \in \Gamma^*$ and $a \in D$. It is inductively defined as follows.

- $q \xrightarrow{\epsilon\,|\,1} q$ for any $q \in Q$.
- $q \xrightarrow{\gamma\,|\,a} q'$ if $a = E(\langle q, \gamma, q'\rangle)$.
- $q \xrightarrow{ww'\,|\,a\otimes b} q'$ if $q \xrightarrow{w\,|\,a} q''$ and $q'' \xrightarrow{w'\,|\,b} q'$.

Then, for two states q and q' and a word w, we consider the total weight of the transitions of the form $q \xrightarrow{w\,|\,a} q'$ defined as follows[2].

$$\delta(q, w, q') = \bigoplus \{a \mid q \xrightarrow{w\,|\,a} q'\}$$

This is well-defined because there are only finitely many transitions of this form and we assume that the semiring is idempotent. In the general theory of weighted automata, we do not impose that the semiring is idempotent [ÉK09]. However, we impose the condition to adopt the simple and intuitive definition above.

3 Stack Signatures

We introduce stack signatures that summarize the effect of a transition on stack as a pair of words over stack alphabet. It is shown that the set of stack signatures forms a monoid, and then a semiring by introducing a partial order on them. Stack signatures naturally appear in the theory of context-free grammars and pushdown systems [Suw09, MT06, TM07]. We adopt the term 'stack signature' introduced by Suwimonteerabuth [Suw09].

The proofs of propositions and theorems in this section are not fundamentally difficult, but require detailed case-analysis. Thus, we have formalized and proved them in Isabelle/HOL by extending our previous work on a formalization of decision procedures on context-free grammars [Min07][3].

[2] This is basically a formal power series, which is used to define the behaviour of weighted automata [ÉK09].

[3] The proof script can be found at http://www.score.cs.tsukuba.ac.jp/~minamide/stacksig.tar.gz

The effect of a transition of a pushdown system can be summarized as a pair of sequences of stack symbols written w_1/w_2 where w_1 are the symbols popped by the transition and w_2 are those pushed by the transition. We consider that pushing γ and then popping the same γ cancel the effect, but popping γ and then pushing γ have the effect γ/γ.

Definition 3. *We call elements of $\Gamma^* \times \Gamma^*$* stack signatures *and write w/w' for a stack signature $\langle w, w' \rangle$.*

– *We say that w_1/w'_1 and w_2/w'_2 are* compatible *if either w'_1 is a prefix of w_2 or w_2 is a prefix of w'_1. Furthermore, they are called* strictly compatible *if $w'_1 = w_2$.*
– *For compatible w_1/w'_1 and w_2/w'_2, we define $w_1/w'_1 \cdot w_2/w'_2$ by*

$$w_1/w'_1 \cdot w_2/w'_2 = \begin{cases} w_1/w'_2 w''_1 & \text{if } w'_1 = w_2 w''_1 \\ w_1 w''_2/w'_2 & \text{if } w_2 = w'_1 w''_2 \end{cases}$$

For example, we have $\gamma_1/\gamma_2 \cdot \gamma_2\gamma_3/\gamma_4 = \gamma_1\gamma_3/\gamma_4$. By introducing an element $\top$ and extending the definition $\cdot$ as follows, $\langle (\Gamma^* \times \Gamma^*) \cup \{\top\}, \cdot, \epsilon/\epsilon \rangle$ forms a monoid. We write $\mathcal{M}_\Gamma$ for this monoid.

$$\begin{aligned} &\top \cdot \sigma = \sigma \cdot \top = \top \quad \text{for } \sigma \in \mathcal{M}_\Gamma \\ &w_1/w'_1 \cdot w_2/w'_2 = \top \text{ if } w_1/w'_1 \text{ and } w_2/w'_2 \text{ are not compatible} \end{aligned}$$

By relaxing the use of terminology, we call an element of $\mathcal{M}_\Gamma$ a *stack signature* and an element of the form w/w' a *proper stack signature.*

The following isomorphism is used to relate automata and pushdown systems. It is clear from $w_1/\epsilon \cdot w_2/\epsilon = w_1w_2/\epsilon$.

Proposition 1. *The set $\{w/\epsilon \mid w \in \Gamma^*\}$ is a submonoid of $\mathcal{M}_\Gamma$. Furthermore, it is isomorphic to Γ^* by the function projecting w from w/ϵ.*

We also introduce a partial order on stack signatures: a transition that pops w_1 and pushes w_2 can be considered as one that pops w_1w and pushes w_2w for any $w \in \Gamma^*$.

Definition 4. *A partial order $\leq$ on stack signatures is defined by $w_1/w_2 \leq w_1w/w_2w$ for $w_1, w_2, w \in \Gamma^*$ and $\sigma \leq \top$ for any stack signature σ.*

It is clear that $(\Gamma^* \times \Gamma^*) \cup \{\top\}$ is a join-semilattice. This partial order is compatible with the binary operation $\cdot$: if $\sigma_1 \leq \sigma'_1$ and $\sigma_2 \leq \sigma'_2$, then $\sigma_1 \cdot \sigma_2 \leq \sigma'_1 \cdot \sigma'_2$. Thus, the monoid of stack signatures is an *ordered monoid*[4].

Furthermore, we can construct an idempotent semiring by introducing the bottom element $\bot$ and extending $\cdot$ for $\bot$ as follows.

$$\bot \cdot x = x \cdot \bot = \bot \qquad \text{for all } x \in (\Gamma^* \times \Gamma^*) \cup \{\top, \bot\}$$

Proposition 2. *Let $S = (\Gamma^* \times \Gamma^*) \cup \{\top, \bot\}$. $\langle S, \sqcup, \cdot, \bot, \epsilon/\epsilon \rangle$ forms an idempotent semiring.*

This semiring is not bounded because $\epsilon/\epsilon \leq \gamma/\gamma \leq \gamma\gamma/\gamma\gamma \leq \cdots$.

[4] A monoid is ordered when it is equipped with a compatible partial order.

4 Semirings Indexed by a Monoid

We introduce semirings indexed by a monoid, which is a typed algebraic structure where a type is an element of a monoid. Weighted pushdown systems are generalized by taking this structure as the domain of weight in the next section.

Definition 5. *Let $\mathcal{M} = \langle M, \cdot, 1_{\mathcal{M}} \rangle$ be a monoid. An indexed semiring $\mathcal{S}$ over $\mathcal{M}$ is a structure $\langle \{D_m\}, \{\oplus_m\}, \{\otimes_{m_1,m_2}\}, \{0_m\}, 1 \rangle$ such that*

- *D_m is a set for each $m \in M$.*
- *$\langle D_m, \oplus_m, 0_m \rangle$ is a commutative monoid for $m \in M$.*
- *$\otimes_{m_1,m_2}$ is an associative binary operation of type $D_{m_1} \times D_{m_2} \to D_{m_1 m_2}$ for $m_1, m_2 \in M$.*

$$(a \otimes_{m_1,m_2} b) \otimes_{m_1 m_2, m_3} c = a \otimes_{m_1, m_2 m_3} (b \otimes_{m_2,m_3} c)$$

- *$1 \in D_{1_{\mathcal{M}}}$ is a neutral element of $\otimes_{m,m'}$: $a \otimes_{m,1_{\mathcal{M}}} 1 = 1 \otimes_{1_{\mathcal{M}},m} a = a$.*
- *$\otimes_{m_1,m_2}$ distributes over $\oplus_m$.*

$$(a \oplus_{m_1} b) \otimes_{m_1,m_2} c = (a \otimes_{m_1,m_2} c) \oplus_{m_1 m_2} (b \otimes_{m_1,m_2} c)$$

$$a \otimes_{m_1,m_2} (b \oplus_{m_2} c) = (a \otimes_{m_1,m_2} b) \oplus_{m_1 m_2} (a \otimes_{m_1,m_2} c)$$

- *0_m is an annihilator with respect to $\otimes_{m,m'}$.*

$$0_{m_1} \otimes_{m_1,m_2} a = 0_{m_1 m_2} = b \otimes_{m_1,m_2} 0_{m_2}$$

We call $\mathcal{S}$ an idempotent indexed semiring if $\mathcal{S}$ is an indexed semiring where $\oplus_m$ is idempotent for all $m \in M$. We introduce partial orders $\sqsubseteq_m$ defined by $a \sqsubseteq_m b$ iff $a \oplus_m b = b$. From distributivity of $\otimes$, it is clear that $\otimes$ is monotonic with respect to $\sqsubseteq_m$.

Proposition 3. *Let $\mathcal{M} = \langle M, \cdot, 1_M \rangle$ be a monoid and $\mathcal{S}$ a semiring indexed by $\mathcal{M}$. If $\mathcal{M}'$ is a submonoid of $\mathcal{M}$, then the restriction of $\mathcal{S}$ on $\mathcal{M}'$ is a semiring indexed by $\mathcal{M}'$.*

The notion of weighted automata can be extended for an indexed semiring over the monoid Γ^* in the straightforward manner.

Definition 6. *Let $\mathcal{S}$ be an idempotent semiring $\langle \{D_w\}, \{\oplus_w\}, \{\otimes_{w_1,w_2}\}, \{0_w\}, 1 \rangle$ indexed by Γ^*. A weighted automaton $\mathcal{A}$ over $\mathcal{S}$ is a structure $\langle \Gamma, Q, E \rangle$ where Q is a finite set of states, and $E : Q \times \Gamma \times Q \to \bigcup_{\gamma \in \Gamma} D_\gamma$ is a set of transition rules assigning a weight such that $E(\langle q, \gamma, q' \rangle) \in D_\gamma$.*

The definition of the transition relation is revised as follows. The only revision is that we apply indexed $\otimes_{w,w'}$ to combine two transitions for w and w'.

- $q \xrightarrow{\epsilon \mid 1} q$ for any $q \in Q$.
- $q \xrightarrow{\gamma \mid a} q'$ if $a = E(\langle q, \gamma, q' \rangle)$.
- $q \xrightarrow{ww' \mid a \otimes_{w,w'} b} q'$ if $q \xrightarrow{w \mid a} q''$ and $q'' \xrightarrow{w' \mid b} q'$.

5 Weighted Pushdown Systems over an Indexed Semiring and Their Reachability Analysis

We introduce weighted pushdown systems over a semiring indexed by the monoid of stack signatures. The reachability analysis of weighted pushdown systems is refined to those over an indexed semiring and the boundedness is relaxed to the local boundedness. We also show that it is possible to construct an ordinary semiring from an indexed semiring, but the obtained semiring is not bounded.

5.1 Weighted Pushdown Systems over an Indexed Semiring

We basically consider pushdown systems over stack alphabet Γ as automata over the monoid of stack signatures $\mathcal{M}_\Gamma$. However, in order to clarify our presentation we introduce the definition of weighted pushdown systems independently.

Definition 7. *Let $\mathcal{S} = \langle \{D_\sigma\}, \{\oplus_\sigma\}, \{\otimes_{\sigma_1,\sigma_2}\}, \{0_\sigma\}, 1\rangle$ be a semiring indexed by $\mathcal{M}_\Gamma$. A weighted pushdown system $\mathcal{P}$ over $\mathcal{S}$ is a structure $\langle P, \Gamma, \Delta\rangle$ where P is a finite set of states, Γ is a stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^* \times \bigcup_{\gamma\in\Gamma, w\in\Gamma^*} D_{\gamma/w}$ is a finite set of transitions such that $a \in D_{\gamma/w}$ for $\langle p, \gamma, p', w, a\rangle \in \Delta$.*

A configuration of pushdown system $\mathcal{P}$ is a pair $\langle p, w\rangle$ where $p \in P$ and $w \in \Gamma^*$. We write $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$ if $\langle p, \gamma, p', w, a\rangle \in \Delta$.

We consider pushdown systems as automata over stack signatures and define the translation relation as follows:

- $p \overset{\epsilon/\epsilon\,|\,1}{\Longrightarrow} p$.
- $p \overset{\gamma/w\,|\,a}{\Longrightarrow} p'$ if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$.
- $p \overset{\sigma_1\cdot\sigma_2\,|\,a}{\Longrightarrow} p'$ if $p \overset{\sigma_1\,|\,a_1}{\Longrightarrow} p'$, $p'' \overset{\sigma_2\,|\,a_2}{\Longrightarrow} p'$, $a = a_1 \otimes_{\sigma_1,\sigma_2} a_2$ and $\sigma_1 \cdot \sigma_2 \neq \top$.

where we have $a \in D_\sigma$ if $p \overset{\sigma\,|\,a}{\Longrightarrow} p'$.

Traditionally, the transition relation on a pushdown system is defined as a relation between configurations. To introduce such a definition, we need to extend an indexed semiring with an additional operation.

Definition 8. *Let $\mathcal{M}$ be an* ordered *monoid with partial order $\leq$. By an indexed semiring over $\mathcal{M}$ we shall mean an indexed semiring $\mathcal{S}$ over $\mathcal{M}$ on which there is a family of conversion functions $\uparrow_{m,m'}: D_m \to D_{m'}$ indexed by pairs of monoid elements $m \leq m'$ such that*

- $\uparrow_{m,m} = \mathrm{id}$.
- $\uparrow_{m,m''} = \uparrow_{m',m''} \circ \uparrow_{m,m'}$ *for all $m \leq m' \leq m''$.*
- $\uparrow_{m,m'}(0_m) = 0_{m'}$ *and* $\uparrow_{m,m'}(a \oplus_m b) = \uparrow_{m,m'}(a) \oplus_{m'} \uparrow_{m,m'}(b)$.
- $\uparrow_{m_1 m_2, m_1' m_2'}(a \otimes_{m_1,m_2} b) = \uparrow_{m_1,m_1'}(a) \otimes_{m_1',m_2'} \uparrow_{m_2,m_2'}(b)$ *for all $m_1 \leq m_1'$ and $m_2 \leq m_2'$.*

For an indexed semiring over the ordered monoid $\mathcal{M}_\Gamma$, we write $\uparrow_w$ for $\uparrow_{w_1/w_2, w_1 w/w_2 w}$ if w_1 and w_2 are clear from the context. Then, the standard definition of the transition relation of a weighted pushdown system is given as follows.

- $\langle p, w\rangle \overset{\uparrow_w(1)}{\Longrightarrow} \langle p, w\rangle$.
- $\langle p, \gamma w'\rangle \overset{\uparrow_{w'}(a)}{\Longrightarrow} \langle p', ww'\rangle$ if $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$.
- $\langle p, w\rangle \overset{a}{\Longrightarrow} \langle p', w'\rangle$ if $\langle p, w\rangle \overset{a_1}{\Longrightarrow} \langle p'', w''\rangle$, $\langle p'', w''\rangle \overset{a_2}{\Longrightarrow} \langle p', w'\rangle$, and $a = a_1 \otimes_{w/w'', w''/w'} a_2$.

Then, these two definitions of transition relations are equivalent in the following sense.

Proposition 4. *If $\langle p, w\rangle \overset{a}{\Longrightarrow} \langle p', w'\rangle$, then there exist σ and a' such that $\sigma \leq w/w'$, $p \overset{\sigma \mid a'}{\Longrightarrow} p'$, and $a = \uparrow_{\sigma, w/w'}(a')$. Conversely, if $p \overset{\sigma \mid a'}{\Longrightarrow} p'$, then $\langle p, w\rangle \overset{\uparrow_{\sigma, w/w'}(a')}{\Longrightarrow} \langle p', w'\rangle$ for all $\sigma \leq w/w'$.*

As a special case of this proposition, we have $\langle p, w\rangle \overset{a}{\Longrightarrow} \langle p', \epsilon\rangle$ iff $p \overset{w/\epsilon \mid a}{\Longrightarrow} p'$.

5.2 Reachability Analysis

We show that the reachability analysis of weighted pushdown systems can be generalized for those over an indexed semiring, where we adopt a localized version of the boundedness of a semiring. We say an indexed idempotent semiring over $\mathcal{M}_\Gamma$ is *locally bounded* if $D_{\gamma/\epsilon}$ is bounded for all $\gamma \in \Gamma$.

First, we focus on the (generalized) backward reachability to a configuration with the empty stack and consider the problem that computes the following function:

$$\delta(p, w, p') = \bigoplus \{a \mid p \overset{w/\epsilon \mid a}{\Longrightarrow} p'\}$$

where the addition above is the extension of $\oplus_{w/\epsilon}$ for a set. This function is well-defined if the indexed semiring is locally bounded. It is clear from the following equation:

$$\delta(p, \gamma w', p') = \bigoplus_{p'' \in P} (\delta(p, \gamma, p'') \otimes_{\gamma/\epsilon, w'/\epsilon} \delta(p'', w', p'))$$

where we have $\delta(p, \gamma, p'') \in D_{\gamma/\epsilon}$ for all $p'' \in P$. Although there are infinitely many transitions of the form $p \overset{\gamma/\epsilon \mid a}{\Longrightarrow} p''$, $\delta(p, \gamma, p'')$ is well-defined because $D_{\gamma/\epsilon}$ is bounded.

We generalize the reachability analysis of weighted pushdown automata for those over an indexed semiring. The algorithm is a generalization of the saturation procedure on $\mathcal{P}$-automata [BEM97, FWW97].

Let us consider a weighted pushdown system $\mathcal{P} = \langle P, \Gamma, \Delta \rangle$ over a semiring $\mathcal{S}$ indexed by $\mathcal{M}_\Gamma$. We apply the procedure to a weighted automaton over the restriction of $\mathcal{S}$ on $\{w/\epsilon \mid w \in \Gamma^*\}$ and start from $\mathcal{A}_0 = \langle P, \Gamma, E_0 \rangle$, which has no transition, *i.e.*, $E_0(\langle p, \gamma, p' \rangle) = 0_{\gamma/\epsilon}$ for $p, p' \in P$ and $\gamma \in \Gamma$. Then, the weighted automaton $\mathcal{A}_{\mathrm{pre}^*}$ representing $\delta_{\mathcal{P}}(p, \gamma, p')$ can be obtained by applying the *standard rule* for weighted pushdown systems to $\mathcal{A}_0$ until saturation. The following is the saturation rule of Reps *et al.* for the backward reachability analysis adapted to our framework [RSJM05].

– If $\langle p, \gamma \rangle \overset{a_1}{\hookrightarrow} \langle p', w \rangle$ and $p' \xrightarrow{w \mid a_2} p''$ in the current automaton, add a transition rule $p \xrightarrow{\gamma \mid a} p''$ where $a = a_1 \otimes_{\gamma/w, w/\epsilon} a_2$.

When we add $p \xrightarrow{\gamma \mid a} p''$, if there already exists transition $p \xrightarrow{\gamma \mid a'} p''$, then we replace it with $p \xrightarrow{\gamma \mid a \oplus_{\gamma/\epsilon} a'} p''$.

Since there is only a finite number of (one-step) transitions in $\mathcal{A}_{\mathrm{pre}^*}$, it is clear that the application of the rule terminates if the indexed semiring is locally bounded.

Theorem 1. *Let $\mathcal{P}$ be a weighted pushdown system over a locally bounded idempotent semiring indexed by $\mathcal{M}_\Gamma$ and $\mathcal{A}_{\mathrm{pre}^*}$ be a weighted automaton obtained by the saturation procedure. Then, we have $p \xrightarrow[\mathcal{A}_{\mathrm{pre}^*}]{\gamma \mid a} p'$ for $a = \delta_{\mathcal{P}}(p, \gamma, p')$.*

As a corollary, we have $p \xrightarrow[\mathcal{A}_{\mathrm{pre}^*}]{w \mid a} p'$ for $a = \delta_{\mathcal{P}}(p, w, p')$. The theorem is proved from the following two lemmas.

Lemma 1. *If $p \overset{w/\epsilon \mid a}{\underset{\mathcal{P}}{\Longrightarrow}} p'$, then $p \xrightarrow[\mathcal{A}_{\mathrm{pre}^*}]{w \mid a'} p'$ and $a \sqsubseteq_{w/\epsilon} a'$ for some a'.*

Let $\mathcal{A}_{i+1}$ be a weighted automaton obtained by applying the saturation rule once to $\mathcal{A}_i$.

Lemma 2. *If $p \xrightarrow[\mathcal{A}_i]{\gamma \mid a} p'$, then $a \sqsubseteq_{\gamma/\epsilon} \delta_{\mathcal{P}}(p, \gamma, p')$.*

5.3 Reachability to a Regular Set of Configurations

In previous works of the reachability analysis of pushdown systems, it is common to consider the reachability problem to a regular set of configurations. For a weighted pushdown automaton over an indexed semiring, this problem must be generalized for a regular set with weight represented by a weighted automaton.

Let us consider an indexed semiring $\mathcal{S}$ over $\mathcal{M}_\Gamma$ and a weighted pushdown system $\mathcal{P}$ over $\mathcal{S}$. We also consider a weighted automaton $\mathcal{A}$ over the restriction of $\mathcal{S}$ on $\{w/\epsilon \mid w \in \Gamma^*\}$ with the initial states q_0 and the set of final states F. Then, the generalized reachability problem to a regular set of configuration $\{\langle p', w' \rangle \mid w'$ is accepted by $\mathcal{A}\}$ is to compute the following function[5].

[5] For simplicity, we consider the set of configurations whose state is p'. It is easy to extend the discussion for the general case.

$$\delta_{\mathcal{P},\mathcal{A}}(p,w,p') = \bigoplus_{q\in F}\{a \otimes_{\sigma,w'/\epsilon} a' \mid p \underset{\mathcal{P}}{\overset{\sigma\,|\,a}{\Longrightarrow}} p',\ q_0 \xrightarrow[\mathcal{A}]{w'\,|\,a'} q, \text{ and } \sigma\cdot w'/\epsilon = w/\epsilon\}$$

This function can be computed by applying the saturation procedure to the pushdown system $\mathcal{P}'$ obtained by combining $\mathcal{P}$ and $\mathcal{A}$ with the identification of p' and q_0. This corresponds to the saturation procedure using $\mathcal{P}$-automata.

The condition $\sigma\cdot w'/\epsilon = w/\epsilon$ above is equivalent to $\sigma \le w/w'$. Furthermore, if the indexed semiring is equipped with the conversion functions $\uparrow_{\sigma_1,\sigma_2}$, we have the following.

$$= \bigoplus_{q\in F}\{\uparrow_{\sigma,w/w'}(a) \otimes_{w/w',w'/\epsilon} a' \mid p \underset{\mathcal{P}}{\overset{\sigma\,|\,a}{\Longrightarrow}} p',\ q_0 \xrightarrow[\mathcal{A}]{w'\,|\,a'} q, \text{ and } \sigma \le w/w'\}$$

$$= \bigoplus_{q\in F}\{a \otimes_{w/w',w'/\epsilon} a' \mid \ \langle p,w\rangle \underset{\mathcal{P}}{\overset{a}{\Longrightarrow}} \langle p',w'\rangle \text{ and } q_0 \xrightarrow[\mathcal{A}]{w'\,|\,a'} q\}$$

5.4 Constructing a Semiring from an Indexed Semiring over Stack Signatures

We show that an ordinary semiring can be constructed from a semiring indexed by the ordered monoid of stack signatures. However, the semiring obtained by the construction is not bounded in general even for a locally bounded indexed semiring. Thus, the standard framework of the reachability analysis of weighted pushdown systems cannot guarantee termination of the saturation procedure. Although a similar construction appears in [Suw09], the definition of $\oplus$ differs from ours and it fails to satisfy the distributivity of $\otimes$ over $\oplus$.

In this section, we assume that $D_\top$ is a singleton set and $D_\top = \{\bullet\}$.

Theorem 2. *Let $\mathcal{S} = \langle\{D_\sigma\},\{\oplus_\sigma\},\{\otimes_{\sigma_1,\sigma_2}\},\{0_\sigma\},1_{\mathcal{S}},\uparrow_{\sigma,\sigma'}\rangle$ be a semiring indexed by the ordered monoid $\mathcal{M}_\Gamma$ and $D = \bigcup_{\sigma\in\mathcal{M}_\Gamma}\{(\sigma,a) \mid a \in D_\sigma\} \cup \{\bot\}$. Then, $\langle D,\oplus,\otimes,\bot,1\rangle$ defined as follows forms a semiring.*

- *1 is $(\epsilon/\epsilon, 1_{\mathcal{S}})$.*
- *$\oplus$ is defined by $\bot \oplus x = x = x \oplus \bot$ for all $x \in D$ and*

$$(\sigma_1,a)\oplus(\sigma_2,b) = \begin{cases} (\sigma_1, a \oplus_{\sigma_1} \uparrow_{\sigma_2,\sigma_1}(b)) & \text{if } \sigma_2 \le \sigma_1 \\ (\sigma_2, \uparrow_{\sigma_1,\sigma_2}(a) \oplus_{\sigma_2} b) & \text{if } \sigma_1 \le \sigma_2 \\ (\top,\bullet) & \text{otherwise} \end{cases}$$

- *$\otimes$ is defined by $(\sigma_1,a)\otimes(\sigma_2,b) = (\sigma_1\sigma_2, a \otimes_{\sigma_1,\sigma_2} b)$ and $x \otimes \bot = \bot = \bot \otimes x$ for all $x \in D$.*

Suwimonteerabuth did not consider the partial order on stack signatures and defined the addition of the semiring $\oplus'$ in the following manner [Suw09].

$$(\sigma_1,a)\oplus'(\sigma_2,b) = \begin{cases} (\sigma_1, a \oplus_{\sigma_1} b) & \text{if } \sigma_1 = \sigma_2 \\ (\top,\bullet) & \text{otherwise} \end{cases}$$

However, $\otimes$ does not distribute over $\oplus'$, and thus fails to form a semiring.

$$((\epsilon/\epsilon, a) \oplus' (\gamma/\gamma, b)) \otimes (\gamma/\gamma, c) = (\top, \bullet) \otimes (\gamma/\gamma, c) = (\top, \bullet)$$

$$\begin{aligned} &((\epsilon/\epsilon, a) \otimes (\gamma/\gamma, c)) \oplus' ((\gamma/\gamma, b) \otimes (\gamma/\gamma, c)) \\ &\qquad = (\gamma/\gamma, a \otimes_{\epsilon/\epsilon,\gamma/\gamma} c) \oplus' (\gamma/\gamma, b \otimes_{\gamma/\gamma,\gamma/\gamma} c) \\ &\qquad = (\gamma/\gamma, a \otimes_{\epsilon/\epsilon,\gamma/\gamma} c \oplus_{\gamma/\gamma} b \otimes_{\gamma/\gamma,\gamma/\gamma} c) \end{aligned}$$

It should be noted that the semiring constructed in Theorem 2 is not bounded as the following sequence shows.

$$(\epsilon/\epsilon, a) \sqsubset (\gamma/\gamma, \uparrow_\gamma(a)) \sqsubset (\gamma\gamma/\gamma\gamma, \uparrow_{\gamma\gamma}(a)) \sqsubset \cdots$$

This is one of the reasons why we refine the formulation of the reachability analysis of weighted pushdown systems in this paper.

6 Simplified Structure: Multiplication on Strictly Compatible Signatures

An indexed semiring has a multiplication indexed by two stack signatures. However, it is often simpler to consider and implement a restricted multiplication defined only for strictly compatible signatures. We show that an indexed semiring over the ordered monoid of stack signatures can be constructed from such a structure.

We introduce *weight structures* that have a restricted multiplication $\odot_{\sigma_1,\sigma_2}$ for strictly compatible σ_1 and σ_2.

Definition 9 (Weight Structure). *A weight structure $\mathcal{W}$ over stack alphabet Γ is $\langle \{D_\sigma\}, \{\oplus_\sigma\}, \{\odot_{\sigma_1,\sigma_2}\}, \{0_\sigma\}, \{1_w\}, \{\uparrow_{\sigma,\sigma'}\}\rangle$ such that*

- *D_σ is a set for each proper stack signature σ.*
- *$\langle D_\sigma, \oplus_\sigma, 0_\sigma\rangle$ is a commutative monoid for proper stack signature σ.*
- *$\odot_{\sigma_1,\sigma_2}$ is an associative binary operation of $D_{\sigma_1} \times D_{\sigma_2} \to D_{\sigma_1\sigma_2}$ for strictly compatible signatures σ_1 and σ_2.*
- *$1_w \in D_{w/w}$ is an indexed unit of $\odot_{\sigma_1,\sigma_2}$: $a \odot_{w'/w,w/w} 1_w = a$ and $1_w \odot_{w/w,w/w'} b = b$.*
- *0_σ is an annihilator with respect to $\odot_{\sigma,\sigma'}$: $0_{\sigma_1} \odot_{\sigma_1,\sigma_2} a = 0_{\sigma_1\sigma_2} = b \odot_{\sigma_1,\sigma_2} 0_{\sigma_2}$.*
- *$\odot$ distributes over $\oplus$.*

$$\begin{aligned} (a \oplus_{\sigma_1} b) \odot_{\sigma_1,\sigma_2} c &= (a \odot_{\sigma_1,\sigma_2} c) \oplus_{\sigma_1\sigma_2} (b \odot_{\sigma_1,\sigma_2} c) \\ a \odot_{\sigma_1,\sigma_2} (b \oplus_{\sigma_2} c) &= (a \odot_{\sigma_1,\sigma_2} b) \oplus_{\sigma_1\sigma_2} (a \odot_{\sigma_1,\sigma_2} c) \end{aligned}$$

- *$\uparrow_{\sigma,\sigma'}$ is a conversion function of $D_\sigma \to D_{\sigma'}$ for $\sigma \le \sigma'$ such that*
  - $\uparrow_{\sigma,\sigma} = \mathrm{id}$ *and* $\uparrow_{\sigma,\sigma''} = \uparrow_{\sigma',\sigma''} \circ \uparrow_{\sigma,\sigma'}$ *for all* $\sigma \le \sigma' \le \sigma''$.
  - $\uparrow_{\sigma,\sigma'}(0_\sigma) = 0_{\sigma'}$ *and* $\uparrow_{\sigma,\sigma'}(a \oplus b) = \uparrow_{\sigma,\sigma'}(a) \oplus \uparrow_{\sigma,\sigma'}(b)$
  - $\uparrow_{w_1/w_2,w_1w'/w_2w'}(a \odot b) = \uparrow_{w_1/w,w_1w'/ww'}(a) \odot \uparrow_{w/w_2,ww'/w_2w'}(b)$
  - $\uparrow_{w/w,ww'/ww'}(1_w) = 1_{ww'}$

We show that the multiplication of an indexed semiring over $\mathcal{M}_\Gamma$ can be obtained from that of a weight structure. Let $\{D'_\sigma\}$ be a family of $\{D_\sigma\} \cup \{D_\top\}$ where $D_\top = \{\bullet\}$. Then, the multiplication on D'_σ is defined as follows.

$$x \otimes_{\sigma_1,\sigma_2} y = \begin{cases} \uparrow_{\sigma_1,\sigma'_1}(x) \odot_{\sigma'_1,\sigma_2} y & \text{if } \sigma_1 \leq \sigma'_1 \text{ and } \sigma'_1 \text{ is strictly compatible with } \sigma_2 \\ x \odot_{\sigma_1,\sigma'_2} \uparrow_{\sigma_2,\sigma'_2}(y) & \text{if } \sigma_2 \leq \sigma'_2 \text{ and } \sigma_1 \text{ is strictly compatible with } \sigma'_2 \\ \bullet & \text{otherwise} \end{cases}$$

The other operations are extended for $D_\top$ in a straightforward manner. Then, we obtain a semiring indexed by the ordered monoid $\mathcal{M}_\Gamma$.

Theorem 3. *Let $\langle \{D_\sigma\}, \{\oplus_\sigma\}, \{\odot_{\sigma_1,\sigma_2}\}, \{0_\sigma\}, \{1_w\}, \{\uparrow_{\sigma,\sigma'}\}\rangle$ be a weight structure. Then, $\langle \{D'_\sigma\}, \{\oplus_\sigma\}, \{\otimes_{\sigma_1,\sigma_2}\}, \{0_\sigma\}, 1_\epsilon, \{\uparrow_{\sigma,\sigma'}\}\rangle$ is an indexed semiring over an ordered monoid $\mathcal{M}_\Gamma$.*

7 Applications

7.1 Encoding of Local Variables into Weight

Suwimonteerabuth applied a semiring similar to one constructed from an indexed semiring to encode local variables of a recursive program into weight [Suw09]. Although his implementation worked without any problem, it is actually not in the standard framework of weighted pushdown systems because the semiring is not bounded.

We show that his encoding can be formulated more naturally with an indexed semiring. In order to simplify our presentation, we give an encoding of a pushdown system into a weighted pushdown system with a singleton stack alphabet. Since local variables can be encoded into stack alphabet, the same approach can be applied for the encoding of local variables.

Let us consider a singleton stack alphabet $\Gamma' = \{\#\}$. We write m/n for a stack signature $\#^m/\#^n$. We will construct a weight structure to translate pushdown systems over stack alphabet Γ. We define weight structure $\mathcal{W}_\Gamma = \langle \{D_\sigma\}, \{\oplus_\sigma\}, \{\odot_{\sigma_1,\sigma_2}\}, \{0_\sigma\}, \{1_w\}, \{\uparrow_{\sigma_1,\sigma_2}\}\rangle$ as follows.

- $D_{m/n}$ is the set of relations over Γ^m and Γ^n: $D_{m/n} = 2^{\Gamma^m \times \Gamma^n}$.
- $0_{m/n} = \emptyset$ and $1_m = \{(x,x) \mid x \in \Gamma^m\}$.
- $R_1 \odot_{l/m,m/n} R_2$ is a composition of relations: $R_1 \circ R_2$ where $R_1 \subseteq \Gamma^l \times \Gamma^m$ and $R_2 \subseteq \Gamma^m \times \Gamma^n$.
- $R_1 \oplus_{m/n} R_2$ is the union of two relations R_1 and R_2: $R_1 \cup R_2$ where $R_1, R_2 \subseteq \Gamma^m \times \Gamma^n$.
- $\uparrow_{l/m,l+1/m+1}$ extends the domain of a relation and is defined by

$$\uparrow_{l+1/m+1}(R) = \{((x,z),(y,z)) \mid (x,y) \in R \wedge z \in \Gamma\}$$

  where we consider $\Gamma^{k+1} = \Gamma^k \times \Gamma$.

It is straightforward to show this structure forms a weight structure. Furthermore, it induces a locally bounded indexed semiring because $D_{m/n}$ is the power set of a finite set and ordered by the set inclusion.

We show how to simulate a pushdown system $\mathcal{P} = \langle P, \Gamma, \Delta\rangle$ by a weighted pushdown system $\mathcal{P}'$ over the weight structure $\mathcal{W}_\Gamma$. Let $\mathcal{P}'$ be $\langle P, \Gamma', \Delta'\rangle$ such that

$$(q, \#, q', \#^m, a) \in \Delta' \qquad \text{iff} \qquad (q, \gamma, q', w) \in \Delta$$

where $|w| = m$ and $a = \{(\gamma, w)\}$. Then, $\mathcal{P}$ and $\mathcal{P}'$ are equivalent in the following sense:

$$p \overset{w/w'}{\underset{\mathcal{P}}{\Longrightarrow}} p' \qquad \Longleftrightarrow \qquad p \overset{m/m' \mid a}{\underset{\mathcal{P}'}{\Longrightarrow}} p' \wedge (w, w') \in a$$

where $m = |w|$ and $m' = |w'|$. Then, we can check the reachability in $\mathcal{P}$ by checking that in $\mathcal{P}'$.

7.2 Reachability Analysis of Conditional Pushdown Systems

Esparza *et al.* introduced pushdown systems with checkpoints that have the ability to inspect the whole stack contents against a regular language [EKS03]. Li and Ogawa reformulated their definition and called them conditional pushdown systems [LO10]. We review conditional pushdown systems and then formulate the reachability analysis in our previous work [MM12] as that of weighted pushdown systems.

Definition 10. *A conditional pushdown system $\mathcal{P}$ is a structure $\langle P, \Gamma, \Delta\rangle$ where P is a finite set of states, Γ is a stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^* \times \mathrm{Reg}(\Gamma)$ is a set of transitions where $\mathrm{Reg}(\Gamma)$ is the set of regular languages over Γ.*

We write $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', w\rangle$ if $\langle p, \gamma, p', w, R\rangle \in \Delta$ as weighted pushdown systems. The transition relation of a conditional pushdown system is defined as follows.

- $\langle p, w\rangle \Longrightarrow \langle p, w\rangle$.
- $\langle p, \gamma w'\rangle \Longrightarrow \langle p', ww'\rangle$ if $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', w\rangle$ and $w' \in R$.
- $\langle p, w\rangle \Longrightarrow \langle p', w'\rangle$ if $\langle p, w\rangle \Longrightarrow \langle p'', w''\rangle$ and $\langle p'', w''\rangle \Longrightarrow \langle p', w'\rangle$.

In the second case above, the transition can be taken only when the current stack contents excluding its top is included in the regular language R given as the condition of the rule.

We show that the transition of a conditional pushdown system can be simulated by that of a weighted pushdown system without conditional rules. Let us design a weight structure for this simulation: we use the same domain for all proper stack signatures σ: $D_\sigma = 2^{\Gamma^*}$. Then, the weight structure $\langle \{D_\sigma\}, \{\oplus_\sigma\}, \{\odot_{\sigma_1,\sigma_2}\}, \{0_\sigma\}, \{1_w\}, \{\uparrow_{\sigma,\sigma'}\}\rangle$ is given as follows.

- $0_\sigma = \emptyset$ and $1_w = \Gamma^*$.
- $a \oplus_\sigma b = a \cup b$.

- $a \odot_{\sigma_1,\sigma_2} b = a \cap b$ for strictly compatible signatures σ_1 and σ_2.
- $\uparrow_{w_1/w_2, w_1 w/w_2 w}(a) = w^{-1}a$ where $w^{-1}a$ is left quotient defined by $w^{-1}a = \{w' \mid ww' \in a\}$.

It is clear that this structure is a weight structure from the basic properties of left quotient and set operations. Then, for a conditional pushdown system $\mathcal{P}$ we obtain a weighted pushdown system $\mathcal{P}'$ over the indexed semiring above by considering a conditional transition rule $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', w\rangle$ as a weighted one.

A conditional pushdown system $\mathcal{P}$ is simulated by a weighted pushdown system $\mathcal{P}'$ in the following sense.

- If $\langle p_1, w_1\rangle \underset{\mathcal{P}}{\Longrightarrow} \langle p_2, w_2\rangle$, then there exist w and σ such that $p_1 \overset{\sigma \mid a}{\underset{\mathcal{P}'}{\Longrightarrow}} p_2$, $w \in a$, and $\uparrow_w(\sigma) = w_1/w_2$.
- If $p_1 \overset{w_1/w_2 \mid a}{\underset{\mathcal{P}'}{\Longrightarrow}} p_2$ and $w \in a$, $\langle p_1, w_1 w\rangle \underset{\mathcal{P}}{\Longrightarrow} \langle p_2, w_2 w\rangle$.

Please note that this weight structure is not locally bounded because 2^{Γ^*} is not bounded with respect to the set inclusion. However, D_σ can be restricted to the set $D \subseteq 2^{\Gamma^*}$ inductively defined as follows.

- $\emptyset \in D$ and $\Gamma^* \in D$.
- $R \in D$ if $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', w\rangle$ for some p, γ, p', w.
- $R_1 \cap R_2 \in D$ and $R_1 \cup R_2 \in D$ if $R_1 \in D$ and $R_2 \in D$.
- $w^{-1}R \in D$ if $R \in D$ and $w \in \Gamma^*$.

This set D is finite because the set of transitions is finite, there are finitely many languages obtained from each regular language with left quotient, and left quotient distributes over union and intersection. Thus, we obtain a locally bounded indexed semiring by using D. This gives the algorithm of the backward reachability analysis for conditional pushdown systems that we used to analyse the HTML5 parser specification [MM12].

8 Related Work

An automaton over a monoid M is called a generalized M-automaton by Eilenberg [Eil74]. The textbook of Sakarovitch discusses automata over several classes of monoids including free groups and commutative monoids [Sak09]. As far as we know, this paper is the first work that discusses the reachability analysis of pushdown systems by considering them as automata over the monoid of stack signatures.

Let us consider a paired alphabet $\widetilde{\Gamma} = \Gamma \cup \overline{\Gamma}$ where $\overline{\Gamma} = \{\overline{a} \mid a \in \Gamma\}$. Letters γ and $\overline{\gamma}$ correspond to a push and a pop of γ, respectively. Then, the monoid $\mathcal{M}_\Gamma$ is closely related to the monoid over $\widetilde{\Gamma}^*$ obtained by Shamir congruence [Sha67], which is generated by $\gamma\overline{\gamma} = \epsilon$. If we add the relation $\gamma\overline{\gamma'} = \top$ for $\gamma \neq \gamma'$, then the reduced form of a word over $\widetilde{\Gamma}$ has the following form: $\overline{w_1}w_2$ or $\top$. If we write $w_1/{w_2}^R$ for $\overline{w_1}w_2$, we obtain a stack signature[6].

[6] ${w_2}^R$ is the reverse of w_2.

Esparza *et al.* showed that conditional pushdown systems can be translated to ordinary pushdown systems [EKS03]. Hence, the reachability can be decided via the translation. However, it is not practical to apply the translation because of exponential blowup of the size of pushdown systems. The algorithm formulated in Section 7.2 as the reachability analysis of weighted pushdown systems has also an exponential complexity. However, it avoids the exponential blowup by the translation before applying the reachability analysis and worked well for the analysis of the HTML5 parser specification.

9 Conclusions

We have introduced the monoid of stack signatures to treat pushdown systems as automata over the monoid. Then, weighted pushdown systems are generalized by adopting a semiring indexed by stack signatures as weight. This generalization makes it possible to relax the restriction of boundedness and extend the applications of the reachability analysis of weighted pushdown systems.

The indexed semirings for the two applications in this paper are given through weight structures. We consider that it is simpler to construct and implement indexed semirings through weight structures than to directly construct them. However, we are not completely satisfied with the formulation of weight structures because their definition looks rather ad hoc mathematically. We would like to investigate more abstract notion corresponding to weight structures.

Acknowledgments. I would like to thank Stefan Schwoon for the inspiring discussions and suggestions. Schwoon informed me of the work of Suwimonteerabuth on the encoding of local variables into weight. This work has been partially supported by JSPS Grant-in-Aid for Science Research (C) 24500028 and the Kayamori Foundation of Informational Science Advancement.

References

[BEM97] 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)

[Eil74] Eilenberg, S.: Automata, Languages, and Machines, vol. A. Academic Press (1974)

[ÉK09] Ésik, Z., Kuich, W.: Finite automata. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata, ch. 3, pp. 69–104. Springer (2009)

[EKS03] Esparza, J., Kucera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Information and Computation 186(2), 355–376 (2003)

[FWW97] Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. In: INFINITY 1997. ENTCS, vol. 9, pp. 27–39 (1997)

[LO10] Li, X., Ogawa, M.: Conditional weighted pushdown systems and applications. In: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 141–150 (2010)

[Min07] Minamide, Y.: Verified Decision Procedures on Context-Free Grammars. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 173–188. Springer, Heidelberg (2007)

[MM12] Minamide, Y., Mori, S.: Reachability Analysis of the HTML5 Parser Specification and Its Application to Compatibility Testing. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 293–307. Springer, Heidelberg (2012)

[MT06] Minamide, Y., Tozawa, A.: XML Validation for Context-Free Grammars. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 357–373. Springer, Heidelberg (2006)

[RSJM05] Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58, 206–263 (2005)

[Sak09] Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press (2009)

[Sha67] Shamir, E.: A representation theorem for algebraic and context-free power series in non commuting variables. Information and Control 11(1/2), 239–254 (1967)

[Suw09] Suwimonteerabuth, D.: Reachability in Pushdown Systems: Algorithms and Applications. PhD thesis, Technischen Universität München (2009)

[TM07] Tozawa, A., Minamide, Y.: Complexity Results on Balanced Context-Free Languages. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 346–360. Springer, Heidelberg (2007)

Underapproximation of Procedure Summaries for Integer Programs*

Pierre Ganty[1], Radu Iosif[2], and Filip Konečný[2,3]

[1] IMDEA Software Institute, Madrid, Spain
[2] VERIMAG/CNRS, Grenoble, France
[3] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. We show how to underapproximate the procedure summaries of recursive programs over the integers using off-the-shelf analyzers for non-recursive programs. The novelty of our approach is that the non-recursive program we compute may capture unboundedly many behaviors of the original recursive program for which stack usage cannot be bounded. Moreover, we identify a class of recursive programs on which our method terminates and returns the precise summary relations without underapproximation. Doing so, we generalize a similar result for non-recursive programs to the recursive case. Finally, we present experimental results of an implementation of our method applied on a number of examples.

1 Introduction

Procedure summaries are relations between the input and return values of a procedure, resulting from its terminating executions. Computing summaries is important, as they are a key enabler for the development of modular verification techniques for inter-procedural programs, such as checking safety, termination or equivalence properties. Summary computation is, however, challenging in the presence of *recursive procedures* with integer parameters, return values, and local variables. While many analysis tools exist for non-recursive programs, only a few ones address the problem of recursion.

In this paper, we propose a novel technique to generate arbitrarily precise *underapproximations* of summary relations. Our technique is based on the following idea. The control flow of procedural programs is captured precisely by the language of a context-free grammar. A k-index underapproximation of this language (where $k \geqslant 1$) is obtained by filtering out those derivations of the grammar that exceed a budget, called *index*, on the number (at most k) of occurrences of non-terminals occurring at each derivation step. As expected, the higher the index, the more complete the coverage of the underapproximation. From there we define the k-index summary relations of a program by considering the k-index underapproximation of its control flow.

Our method then reduces the computation of k-index summary relations for a recursive program to the computation of summary relations for a non-recursive program, which is, in general, easier to compute because of the absence of recursion. The reduction was inspired by a decidability proof [4] in the context of Petri nets.

* Supported by the French National Project ANR-09-SEGI-016 VERIDYC, the Spanish Ministry of Economy and Finance grant (TIN2010-20639), and the Rich Model Toolkit initiative.

The contributions of this paper are threefold. First, we show that, for a given index, recursive programs can be analyzed using off-the-shelf analyzers designed for non-recursive programs. Second, we identify a class of recursive programs, with possibly unbounded stack usage, on which our technique is complete i.e., it terminates and returns the precise result. Third, we present experimental results of an implementation of our method applied on a number of examples.

Related Work. programs handling integers (in general, unbounded data domains) has gained significant interest with the seminal work of Sharir and Pnueli [21]. They proposed two approaches for interprocedural dataflow analysis. The first one keeps precise values (*call strings*) up to a limited depth of the recursion stack, which bounds the number of executions. In contrast to the methods based on the call strings approach, our method can also analyse precisely certain programs for which the stack is unbounded, allowing for unbounded number of executions to be represented at once.

The second approach of Sharir and Pnueli is based on computing the least fixed point of a system of recursive dataflow equations (the *functional approach*). This approach to interprocedural analysis is based on computing an increasing *Kleene sequence* of abstract summaries. It is to be noticed that abstraction is key to ensuring termination of the Kleene sequence, the result being an over-approximation of the precise summary. Recently [10], the *Newton sequence* was shown to converge at least as fast as the Kleene sequence. The intuition behind the Newton sequence is to consider control paths in the program of increasing *index*. Our contribution can be seen as a technique to compute the iterates of the Newton sequence for programs with integer parameters, return values, and local variables, the result being, at each step, an under-approximation of the precise summary.

The complexity of the functional approach was shown to be polynomial in the size of the (finite) abstract domain, in the work of Reps, Horwitz and Sagiv [20]. This result is achieved by computing summary information, in order to reuse previously computed information during the analysis. Following up on this line of work, most existing abstract analyzers, such as INTERPROC [17], also use relational domains to compute *overapproximations* of function summaries – typically widening operators are used to ensure termination of fixed point computations. The main difference of our method with respect to static analyses is the use of underapproximation instead of overapproximation. If the final purpose of the analysis is program verification, our method will not return false positives. Moreover, the coverage can be increased by increasing the bound on the derivation index.

Previous works have applied model checking based on abstraction refinement to recursive programs. One such method, known as *nested interpolants* represents programs as nested word automata [3], which have the same expressive power as the visibly pushdown grammars used in our paper. Also based on interpolation is the WHALE algorithm [2], which combines partial exploration of the execution paths (underapproximation) with the overapproximation provided by a predicate-based abstract post operator, in order to compute summaries that are sufficient to prove a given safety property. Another technique, similar to WHALE, although not handling recursion, is the SMASH algorithm [13] which combines may- and must-summaries for compositional verification of safety properties. These approaches are, however, different in spirit from ours, as their goal is proving given safety properties of programs, as opposed to computing the summaries of procedures independently of their calling context, which is our case. We argue that summary computation can be applied beyond safety checking, e.g., to prove termination [5], or program equivalence.

2 Preliminaries

Grammars. A *context-free grammar* (or simply grammar) is a tuple $G = (\mathcal{X}, \Sigma, \delta)$ where $\mathcal{X}$ is a finite nonempty set of *nonterminals*, Σ is a finite nonempty *alphabet* and $\delta \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^*$ is a finite set of *productions*. The production (X, w) may also be noted $X \rightarrow w$. Also define $head(X \rightarrow w) = X$ and $tail(X \rightarrow w) = w$. Given two strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we define a *step* $u \Longrightarrow v$ if there exists a production $(X, w) \in \delta$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\Longrightarrow^*$ to denote the reflexive transitive closure of $\Longrightarrow$. The *language* of G produced by a nonterminal $X \in \mathcal{X}$ is the set $L_X(G) = \{w \in \Sigma^* \mid X \Longrightarrow^* w\}$ and we call any sequence of steps from a nonterminal X to $w \in \Sigma^*$ a *derivation* from X. Given $X \Longrightarrow^* w$, we call the sequence $\gamma \in \delta^*$ of productions used in the derivation a *control word* and write $X \overset{\gamma}{\Longrightarrow} w$ to denote that the derivation conforms to γ.

Visibly Pushdown Grammars. To model the control flow of procedural programs we use languages generated by visibly pushdown grammars, a subset of context-free grammars. In this setting, words are defined over a *tagged alphabet* $\widehat{\Sigma} = \Sigma \cup \langle\Sigma \cup \Sigma\rangle$, where $\langle\Sigma = \{\langle a \mid a \in \Sigma\}$ represents procedure *call* sites and $\Sigma\rangle = \{a\rangle \mid a \in \Sigma\}$ represents procedure *return* sites. Formally, a *visibly pushdown grammar* $G = (\mathcal{X}, \widehat{\Sigma}, \delta)$ is a grammar that has only productions of the following forms, for some $a, b \in \Sigma$:

$$X \rightarrow a \qquad\qquad X \rightarrow a\,Y \qquad\qquad X \rightarrow \langle a\,Y\,b\rangle\,Z$$

It is worth pointing that, for our purposes, we do not need a visibly pushdown grammar to generate the empty string ε. Each tagged word generated by visibly pushdown grammars is associated a *nested word* [3] the definition of which we briefly recall. Given a finite alphabet Σ, a *nested word* over Σ is a pair $(w, \rightsquigarrow)$, where $w = a_1 a_2 \ldots a_n \in \Sigma^*$, and $\rightsquigarrow \subseteq \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ is a set of *nesting edges* (or simply edges) where:

1. $i \rightsquigarrow j$ only if $i < j$, i.e. edges only go forward;
2. $|\{j \mid i \rightsquigarrow j\}| \leqslant 1$ and $|\{i \mid i \rightsquigarrow j\}| \leqslant 1$, i.e. no two edges share a call/return position;
3. if $i \rightsquigarrow j$ and $k \rightsquigarrow \ell$ then it is not the case that $i < k \leqslant j < \ell$, i.e. edges do not cross.

Intuitively, we associate a nested word to a tagged word as follows: there is an edge between tagged symbols $\langle a$ and $b\rangle$ iff both are generated at the same derivation step. For instance looking forward at Ex. 2 consider the tagged word $w = \tau_1\tau_2\langle\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\rangle\tau_4$ resulting from a derivation $Q_1^{init} \Rightarrow^* w$. The nested word associated to w is $(\tau_1\tau_2\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\tau_4, \{3 \rightsquigarrow 8\})$. Finally, let w_nw denote the mapping which given a tagged word in the language of a visibly pushdown grammar returns the nested word thereof.

Integer Relations. We denote by $\mathbb{Z}$ the set of integers. Let $\mathbf{x} = \{x_1, x_2, \ldots, x_d\}$ be a set of variables for some $d > 0$. Define $\mathbf{x}'$ the *primed* variables of $\mathbf{x}$ to be $\{x'_1, x'_2, \ldots, x'_d\}$. All variables range over $\mathbb{Z}$. We denote by $\overrightarrow{\mathbf{y}}$ an ordered sequence $\langle y_1, \ldots, y_k\rangle$ of variables, and by $|\overrightarrow{\mathbf{y}}|$ its length k. By writing $\overrightarrow{\mathbf{y}} \subseteq \mathbf{x}$ we mean that each variable in $\overrightarrow{\mathbf{y}}$ belongs to $\mathbf{x}$. For sequences $\overrightarrow{\mathbf{y}}$ and $\overrightarrow{\mathbf{z}}$ of length k, $\overrightarrow{\mathbf{y}} = \overrightarrow{\mathbf{z}}$ stands for the equality $\bigwedge_{i=1}^{k} y_i = z_i$.

A *linear term t* is a linear combination of the form $a_0 + \sum_{i=1}^{d} a_i x_i$, where $a_0, \ldots, a_d \in \mathbb{Z}$. An *atomic proposition* is a predicate of the form $t \leqslant 0$, where t is a linear term. We consider formulae in the first-order logic over atomic propositions $t \leqslant 0$, also known as *Presburger arithmetic*. A *valuation* of $\mathbf{x}$ is a function $\nu : \mathbf{x} \to \mathbb{Z}$. The set of all valuations of $\mathbf{x}$ is denoted by $\mathbb{Z}^{\mathbf{x}}$. If $\overrightarrow{\mathbf{y}} = \langle y_1, \ldots, y_k \rangle$ is an ordered sequence of variables, we denote by $\nu(\overrightarrow{\mathbf{y}})$ the sequence of integers $\langle \nu(y_1), \ldots, \nu(y_k) \rangle$. An arithmetic formula $\mathcal{R}(\mathbf{x}, \mathbf{y}')$ defining a relation $R \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{y}}$ is evaluated with respect to two valuations $\nu_1 \in \mathbb{Z}^{\mathbf{x}}$ and $\nu_2 \in \mathbb{Z}^{\mathbf{y}}$, by replacing each $x \in \mathbf{x}$ by $\nu_1(x)$ and each $y' \in \mathbf{y}'$ by $\nu_2(y)$ in $\mathcal{R}$. The composition of two relations $R_1 \subseteq \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{y}}$ and $R_2 \subseteq \mathbb{Z}^{\mathbf{y}} \times \mathbb{Z}^{\mathbf{z}}$ is denoted by $R_1 \circ R_2 = \{(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^{\mathbf{x}} \times \mathbb{Z}^{\mathbf{z}} \mid \exists \mathbf{t} \in \mathbb{Z}^{\mathbf{y}} . (\mathbf{u}, \mathbf{t}) \in R_1 \text{ and } (\mathbf{t}, \mathbf{v}) \in R_2\}$. For a subset $\mathbf{y} \subseteq \mathbf{x}$, we denote $\nu\downarrow_{\mathbf{y}} \in \mathbb{Z}^{\mathbf{y}}$ the projection of ν onto variables $\mathbf{y} \subseteq \mathbf{x}$. Finally, given two valuations $I, O \in \mathbb{Z}^{\mathbf{x}}$, we denote by $I \cdot O$ their concatenation and we define $\mathbb{Z}^{\mathbf{x} \times \mathbf{x}} = \{I \cdot O \mid I, O \in \mathbb{Z}^{\mathbf{x}}\}$.

3 Integer Recursive Programs

We consider in the following that programs are collections of procedures calling each other, possibly according to recursive schemes. Formally, an *integer program* is an indexed tuple $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$, where $P_1, \ldots, P_n$ are *procedures*. Each procedure is a tuple $P_i = \langle \mathbf{x}_i, \overrightarrow{\mathbf{x}}_i^{in}, \overrightarrow{\mathbf{x}}_i^{out}, S_i, q_i^{init}, F_i, \Delta_i \rangle$, where $\mathbf{x}_i$ are the *local* variables[1] of P_i ($\mathbf{x}_i \cap \mathbf{x}_j = \emptyset$ for all $i \neq j$), $\overrightarrow{\mathbf{x}}_i^{in}, \overrightarrow{\mathbf{x}}_i^{out} \subseteq \mathbf{x}_i$ are the ordered tuples of input and output variables, S_i are the *control states* of P_i ($S_i \cap S_j = \emptyset$, for all $i \neq j$), $q_i^{init} \in S_i$ is the *initial*, and $F_i \subseteq S_i$ are the *final* states of P_i, and Δ_i is a set of *transitions* of one of the following forms:

- $q \xrightarrow{\mathcal{R}(\mathbf{x}_i, \mathbf{x}_i')} q'$ is an *internal transition*, where $q, q' \in S_i$, and $\mathcal{R}(\mathbf{x}_i, \mathbf{x}_i')$ is a Presburger arithmetic relation involving only the local variables of P_i;
- $q \xrightarrow{\overrightarrow{\mathbf{z}}' = P_j(\overrightarrow{\mathbf{u}})} q'$ is a *call*, where $q, q' \in S_i$, P_j is the callee, $\overrightarrow{\mathbf{u}}$ are linear terms over $\mathbf{x}_i$, $\overrightarrow{\mathbf{z}} \subseteq \mathbf{x}_i$ are variables, such that $|\overrightarrow{\mathbf{u}}| = |\overrightarrow{\mathbf{x}}_j^{in}|$ and $|\overrightarrow{\mathbf{z}}| = |\overrightarrow{\mathbf{x}}_j^{out}|$.

We define the *size of the program* $\sharp\mathcal{P} = \sum_{i=1}^{n} |\Delta_i|$ to be the total number of transition rules, and $loc(\mathcal{P}) = \sum_{i=1}^{n} |S_i|$ be the number of control locations in $\mathcal{P}$. The *call graph* of a program $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ is a directed graph with vertices $P_1, \ldots, P_n$ and an edge (P_i, P_j), for each P_i and P_j, such that P_i has a call to P_j. A program is said to be *recursive* if its call graph has at least one cycle, and *non-recursive* if its call graph is a dag. Finally, let $n\mathcal{F}(P_i)$ denotes the set $S_i \backslash F_i$ of non-final states of P_i, and $n\mathcal{F}(\mathcal{P}) = \bigcup_{i=1}^{n} n\mathcal{F}(\mathcal{P})$ be the set of non-final states of $\mathcal{P}$.

Simplified Syntax. To ease the description of programs defined in this paper, we use a simplified, human readable, imperative language such that each procedure of the program conforms to the following grammar:[2]

$$P ::= \textbf{proc } P_i(id^*)\textbf{begin var } id^* \; S \textbf{ end} \qquad\qquad S ::= S; S \mid \textbf{assume } f$$
$$S ::= id^n \leftarrow t^n \mid id \leftarrow P_i(t^*) \mid P_i(t^*) \mid \textbf{return } (id + \varepsilon) \mid \textbf{goto } \ell^+ \mid \textbf{havoc } id^+$$

[1] Observe that there are no global variables in the definition of integer program. Those can be encoded as input and output variables to each procedure.

[2] Our simplified syntax does not seek to capture the generality of integer programs. Instead, our goal is to give a convenient notation for the programs given in this paper and only those.

```
       proc  P(x)
        begin
              var z;
              assume x ⩾ 0;
              goto then or
              else;
then:         assume x > 0;
              z ← P(x − 1);
              z ← z + 2;
              return z;
else:         assume x ⩽ 0;
              z ← 0;
              havoc x;
              return z;
        end
```

$$\begin{tikzcd}
 & q_1^{init} \arrow[d, "x\geqslant 0\land x'=x\land z'=z", "\textcircled{t_1}"'] & \\
 & q_2 \arrow[dl, "x>0\land x'=x\land z'=z"', "\textcircled{t_2}"] \arrow[dr, "x\leqslant 0\land x'=x\land z'=z", "\textcircled{t_5}"'] & \\
q_3 \arrow[d, "z'=P(x-1)"', "\textcircled{t_3}"] & & q_5 \arrow[d, "z'=0\land x'=x", "\textcircled{t_6}"'] \\
q_4 \arrow[d, "z'=z+2\land x'=x"', "\textcircled{t_4}"] & & q_6 \arrow[d, "z'=z", "\textcircled{t_7}"'] \\
q'_f & & q''_f
\end{tikzcd}$$

Fig. 1. Example of a simplified imperative program and its integer program thereof

The local variables occurring in P are denoted by id, linear terms by t, Presburger formulae by f, and control labels by ℓ. Each procedure consists in local declarations followed by a sequence of statements. Statements may carry a label. Program statements can be either assume statements[3], (parallel) assignments, procedure calls (possibly with a return value), return to the caller (possibly with a value), non-deterministic jumps **goto** ℓ_1 **or** ... **or** ℓ_n, and **havoc** $x_1, x_2, \ldots, x_n$ statements[4]. We consider the usual syntactic requirements (used variables must be declared, jumps are well defined, no jumps outside procedures, etc.). We do not define them, it suffices to know that all simplified programs in this paper comply with the requirements. A program using the simplified syntax can be easily translated into the formal syntax, as shown at Fig. 1.

Example 1. Figure 1 shows a program in our simplified imperative language and its corresponding integer program $\mathcal{P}$. Formally, $\mathcal{P} = \langle P \rangle$ where P is defined as: $\langle \{x,z\}, \langle x \rangle, \langle z \rangle, \{q_1^{init}, q_2, q_3, q_4, q_5, q_6, q'_f, q''_f\}, q_1^{init}, \{q'_f, q''_f\}, \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\} \rangle$. Since P calls itself (t_3), this program is recursive. ■

Semantics. We are interested in computing the *summary relation* between the values of the input and output variables of a procedure. To this end, we give the semantics of a program $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ as a tuple of relations R_q describing, for each non-final control state $q \in n\mathcal{F}(P_i)$, the effect of the program when started in q upon reaching a state in F_i. An *interprocedurally valid path* is represented by a tagged word over an alphabet $\widehat{\Theta}$, which maps each internal transition t to a symbol τ, and each call transition t to a pair of symbols $\langle \tau, \tau \rangle \in \widehat{\Theta}$. In the sequel, we denote by Q the nonterminal corresponding to the control state q, and by $\tau \in \Theta$ the alphabet symbol corresponding to the transition t of $\mathcal{P}$. Formally, we associate $\mathcal{P}$ a visibly pushdown grammar, denoted in the rest of the paper by $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$, such that $Q \in \mathcal{X}$ if and only if $q \in n\mathcal{F}(\mathcal{P})$ and:

[3] **assume** f is executable if and only if the current values of the variables satisfy f.

[4] **havoc** assigns non deterministically chosen integers to $x_1, x_2, \ldots, x_n$.

(a) $Q \rightarrow \tau \in \delta$ if and only if $t\colon q \xrightarrow{\mathcal{R}} q'$ and $q' \notin n\mathcal{F}(\mathcal{P})$

(b) $Q \rightarrow \tau\ Q' \in \delta$ if and only if $t\colon q \xrightarrow{\mathcal{R}} q'$ and $q' \in n\mathcal{F}(\mathcal{P})$

(c) $Q \rightarrow \langle \tau\ Q_j^{init}\ \tau\rangle\ Q' \in \delta$ if and only if $t\colon q \xrightarrow{\overrightarrow{\mathbf{z}}'=P_j(\overrightarrow{\mathbf{u}})} q'$

It is easily seen that interprocedurally valid paths in $\mathcal{P}$ and tagged words in $G_{\mathcal{P}}$ are in one-to-one correspondence. In fact, each interprocedurally valid path of $\mathcal{P}$ between state $q \in n\mathcal{F}(P_i)$ and a state of F_i, where $1 \leqslant i \leqslant n$, corresponds exactly to one tagged word of $L_Q(G_{\mathcal{P}})$.

Example 2. (continued from Ex. 1) The visibly pushdown grammar $G_{\mathcal{P}}$ corresponding to $\mathcal{P}$ consists of the following variables and labelled productions:

$$\begin{array}{llll}
p_1^b \overset{def}{=} Q_1^{init} \rightarrow \tau_1\ Q_2 & & p_3^c \overset{def}{=} Q_3 \rightarrow \langle \tau_3\ Q_1^{init}\ \tau_3\rangle\ Q_4 \\
p_2^b \overset{def}{=} Q_2 \quad \rightarrow \tau_2\ Q_3 & & p_4^a \overset{def}{=} Q_4 \rightarrow \tau_4 \\
p_5^b \overset{def}{=} Q_2 \quad \rightarrow \tau_5\ Q_5 & & p_6^b \overset{def}{=} Q_5 \rightarrow \tau_6\ Q_6 \\
 & & p_7^a \overset{def}{=} Q_6 \rightarrow \tau_7
\end{array}$$

In the following, we use superscripts a,b,c to distinguish productions of the form $Q \rightarrow \tau$, $Q \rightarrow \tau Q'$ or $Q \rightarrow \langle \tau Q_j^{init} \tau\rangle Q'$, respectively. $L_{Q_1^{init}}(G_{\mathcal{P}})$ includes the word $w = \tau_1\tau_2\langle\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\rangle\tau_4$, of which $w\_nw(w) = (\tau_1\tau_2\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\tau_4, \{3 \rightsquigarrow 8\})$ is the corresponding nested word. The word w corresponds to an interprocedurally valid path where P calls itself once. Let $\gamma_1 = p_1^b p_2^b p_3^c p_4^a p_1^b p_5^b p_6^b p_7^a$ and $\gamma_2 = p_1^b p_2^b p_3^c p_1^b p_5^b p_6^b p_7^a p_4^a$ be two control words such that $Q_1^{init} \xRightarrow{\gamma_1} w$ and $Q_1^{init} \xRightarrow{\gamma_2} w$. ■

The semantics of a program is the union of the semantics of the nested words corresponding to its executions, each of which being a relation over input and output variables. To define the semantics of a nested word, we first associate to each $\tau \in \widehat{\Theta}$ an integer relation ρ_τ, defined as follows:

– for an internal transition $t\colon q \xrightarrow{\mathcal{R}} q' \in \Delta_i$, let $\rho_\tau \equiv \mathcal{R}(\mathbf{x}_i, \mathbf{x}_i') \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$;

– for a call transition $t : q \xrightarrow{\overrightarrow{\mathbf{z}}'=P_j(\overrightarrow{\mathbf{u}})} q' \in \Delta_i$, we define a *call relation* $\rho_{\langle\tau} \equiv (\overrightarrow{\mathbf{x}}_j^{in\prime} = \overrightarrow{\mathbf{u}}) \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_j}$, a *return relation* $\rho_{\tau\rangle} \equiv (\overrightarrow{\mathbf{z}}' = \overrightarrow{\mathbf{x}}_j^{out}) \subseteq \mathbb{Z}^{\mathbf{x}_j} \times \mathbb{Z}^{\mathbf{x}_i}$ and a *frame relation* $\phi_\tau \equiv \bigwedge_{x \in \mathbf{x}_i \setminus \overrightarrow{\mathbf{z}}} x' = x \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$.

We define the semantics of the program $\mathcal{P} = \langle P_1, \ldots, P_n\rangle$ in a top-down manner. Assuming a fixed ordering of the non-final states in the program, i.e. $n\mathcal{F}(\mathcal{P}) = \langle q_1, \ldots, q_m\rangle$, the semantics of the program $\mathcal{P}$, denoted $[\![\mathcal{P}]\!]$, is the tuple of relations $\langle [\![q_1]\!], \ldots, [\![q_m]\!]\rangle$. For each non-final control state $q \in n\mathcal{F}(P_i)$ where $1 \leqslant i \leqslant n$, we denote by $[\![q]\!] \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$ the relation (over the local variables of procedure P_i) defined as $[\![q]\!] = \bigcup_{\alpha \in L_Q(G_{\mathcal{P}})} [\![\alpha]\!]$.

It remains to define $[\![\alpha]\!]$, the semantics of the tagged word α. Out of convenience, we define the semantics of its corresponding nested word $w\_nw(\alpha) = (\tau_1 \ldots \tau_\ell, \rightsquigarrow)$ over alphabet Θ, and define $[\![\alpha]\!] = [\![w\_nw(\alpha)]\!]$. For a nesting relation $\rightsquigarrow\ \subseteq \{1, \ldots, \ell\} \times \{1, \ldots, \ell\}$, we define $\rightsquigarrow_{i,j} = \{(s-(i-1), t-(i-1)) \mid (s,t) \in\ \rightsquigarrow \cap \{i, \ldots, j\} \times \{i, \ldots, j\}\}$, for some $i, j \in \{1, \ldots, \ell\}$, $i < j$. Finally, we define $[\![(\tau_1 \ldots \tau_\ell, \rightsquigarrow)]\!] \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$ (recall that $\alpha \in L_Q(G_{\mathcal{P}})$ and q is a state of P_i) as follows:

$$[\![(\tau_1 \ldots \tau_\ell, \rightsquigarrow)]\!] = \begin{cases} \rho_{\tau_1} & \text{if } \ell = 1 \\ \rho_{\tau_1} \circ [\![(\tau_2 \ldots \tau_\ell, \rightsquigarrow_{2,\ell})]\!] & \text{if } \ell > 1 \text{ and } \forall 1 \leqslant j \leqslant \ell \colon 1 \not\rightsquigarrow j \\ CaRet_\tau \circ [\![(\tau_{j+1} \ldots \tau_\ell, \rightsquigarrow_{j+1,\ell})]\!] & \text{if } \ell > 1 \text{ and } \exists 1 \leqslant j \leqslant \ell \colon 1 \rightsquigarrow j \end{cases}$$

where, in the last case, which corresponds to call transition $t : q \xrightarrow{\vec{\mathbf{z}}' = P_d(\vec{\mathbf{u}})} q' \in \Delta_i$, we have $\tau_1 = \tau_j = \tau$ and define $CaRet_\tau = \big(\rho_{\langle\tau} \circ [\![\tau_2 \ldots \tau_{j-1}, \rightsquigarrow_{2,j-1})]\!] \circ \rho_{\tau\rangle}\big) \cap \phi_\tau$.

Example 3. (continued from Ex. 2) The semantics of a given the nested word $\theta = (\tau_1\tau_2\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\tau_4, \{3 \rightsquigarrow 8\})$ is a relation between valuations of $\{x, z\}$, given by:

$$[\![\theta]\!] = \rho_{\tau_1} \circ \rho_{\tau_2} \circ \big((\rho_{\langle\tau_3} \circ \rho_{\tau_1} \circ \rho_{\tau_5} \circ \rho_{\tau_6} \circ \rho_{\tau_7} \circ \rho_{\tau_3\rangle}) \cap \phi_{t_3}\big) \circ \rho_{\tau_4}$$

One can verify that $[\![\theta]\!] \equiv x = 1 \wedge z' = 2$, i.e. the result of calling P with an input valuation $x = 1$ is the output valuation $z = 2$. ■

Finally, we introduce a few useful notations. By $[\![\mathcal{P}]\!]_q$ we denote the component of $[\![\mathcal{P}]\!]$ corresponding to $q \in n\mathcal{F}(\mathcal{P})$. Slightly abusing notations, we define $L_{P_i}(G_\mathcal{P})$ as $L_{Q_i^{init}}(G_\mathcal{P})$ and $[\![\mathcal{P}]\!]_{P_i}$ as $[\![\mathcal{P}]\!]_{q_i^{init}}$. Finally, define $[\![\mathcal{P}]\!]_{P_i}^{i/o} = \{\langle I{\downarrow}_{x_i^{in}}, O{\downarrow}_{x_i^{out}}\rangle \mid \langle I, O\rangle \in [\![\mathcal{P}]\!]_{P_i}\}$.

4 Underapproximating the Program Semantics

In this section we define a family of underapproximations of $[\![\mathcal{P}]\!]$, called *bounded-index underapproximations*. Then we show that each k-index underapproximation of the semantics of a (possibly recursive) program $\mathcal{P}$ coincides with the semantics of a non-recursive program computable from $\mathcal{P}$ and k. The central notion of bounded-index derivation is introduced in the following followed by basic properties about them.

Definition 1. *Given a grammar $G = (\mathcal{X}, \Sigma, \delta)$ with relation $\Longrightarrow$ between strings, for every $k \geqslant 1$ we define the relation $\overset{(k)}{\Longrightarrow} \subseteq \Longrightarrow$ as follows: $u \overset{(k)}{\Longrightarrow} v$ iff $u \Longrightarrow v$ and both u and v contain at most k occurrences of variables from $\mathcal{X}$. We denote by $\overset{(k)}{\Longrightarrow}{}^\star$ the reflexive transitive closure of $\overset{(k)}{\Longrightarrow}$. Hence given X and k define $L_X^{(k)}(G) = \{w \in \Sigma^* \mid X \overset{(k)}{\Longrightarrow}{}^\star w\}$, and we call the $\overset{(k)}{\Longrightarrow}$-derivation of $w \in \Sigma^*$ from X a k-*index derivation. *A grammar G is said to have* index *k whenever $L_X(G) = L_X^{(k)}(G)$ for each $X \in \mathcal{X}$.*[5]

Lemma 1. *For every grammar the following properties hold: (1) $\overset{(k)}{\Longrightarrow} \subseteq \overset{(k+1)}{\Longrightarrow}$ for all $k \geqslant 1$; (2) $\Longrightarrow = \bigcup_{k=1}^{\infty} \overset{(k)}{\Longrightarrow}$; (3) $BC \overset{(k)}{\Longrightarrow}{}^\star w \in \Sigma^*$ iff there exist w_1, w_2 such that $w = w_1 w_2$ and either (i) $B \overset{(k-1)}{\Longrightarrow}{}^\star w_1$, $C \overset{(k)}{\Longrightarrow}{}^\star w_2$, or (ii) $C \overset{(k-1)}{\Longrightarrow}{}^\star w_2$ and $B \overset{(k)}{\Longrightarrow}{}^\star w_1$.*

The main intuition behind our method is to filter out interprocedurally valid paths which can not be produced by k-index derivations. Our analysis is then carried out on the remaining paths produced by k-index derivations only.

[5] Gruska [15] proved that deciding whether $L_X(G) = L_X^{(k)}(G)$ for some $k \geqslant 1$ is undecidable.

Example 4. (continued form Ex. 2) P is a (non-tail) recursive procedure and $G_{\mathcal{P}}$ models its control flow. Inspecting $G_{\mathcal{P}}$ reveals that $L_{Q_1^{init}}(G_{\mathcal{P}}) = \{(\tau_1\tau_2\langle\tau_3)^n\tau_1\tau_5\tau_6\tau_7(\tau_3\rangle\tau_4)^n \mid n \geqslant 0\}$. For each value of n we give a 2-index derivation capturing the word: repeat n times the steps $Q_1^{init} \overset{p_1^b p_2^b p_3^c}{\Longrightarrow} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle Q_4 \overset{p_4^a}{\Longrightarrow} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle\tau_4$ followed by the steps $Q_1^{init} \overset{p_1^b p_5^b p_6^b p_7^a}{\Longrightarrow} \tau_1\,\tau_5\,\tau_6\,\tau_7$. Therefore the 2-index approximation of $G_{\mathcal{P}}$ shows that $L_{Q_1^{init}}(G_{\mathcal{P}}) = L^{(2)}_{Q_1^{init}}(G_{\mathcal{P}})$. However bounding the number of times P calls itself up to 2 results in 3 interprocedurally valid paths (for $n = 0,1,2$). □

Given $k \geqslant 1$, we define the *k-index semantics* of $\mathcal{P}$ as $[\![\mathcal{P}]\!]^{(k)} = \langle [\![q_1]\!]^{(k)}, \ldots, [\![q_m]\!]^{(k)}\rangle$, where the k-index semantics of a non-final control state q of a procedure P_i is the relation $[\![q]\!]^{(k)} \subseteq \mathbb{Z}^{\mathbf{x}_i} \times \mathbb{Z}^{\mathbf{x}_i}$, defined as $[\![q]\!] = \bigcup_{\alpha \in L_Q^{(k)}(G_{\mathcal{P}})} [\![\alpha]\!]$.

4.1 Computing Bounded-Index Underapproximations

In what follows, we define a source-to-source transformation that takes in input a recursive program $\mathcal{P}$, an integer $k \geqslant 1$ and returns a *non-recursive* program $\mathcal{H}^k$ which has the same semantics as $[\![\mathcal{P}]\!]^{(k)}$ (modulo projection on some variables). Therefore every off-the-shelf tool, that computes the summary semantics for a non-recursive program, can be used to compute the k-index semantics of $\mathcal{P}$, for any given $k \geqslant 1$.

Let $\mathcal{P} = \langle P_1, \ldots, P_n\rangle$ be a program, and $\mathbf{x} = \bigcup_{i=1}^n \mathbf{x}_i$ be the set of all variables in $\mathcal{P}$. As we did previously, we assume a fixed ordering $\langle q_1, \ldots, q_m\rangle$ on the set $n\mathcal{F}(\mathcal{P})$. Let $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$ be the visibly pushdown grammar associated with $\mathcal{P}$, such that each non-final state q of $\mathcal{P}$ is associated a nonterminal $Q \in \mathcal{X}$. Then we define a *non-recursive* program $\mathcal{H}^K$ that captures the K-index semantics of $\mathcal{P}$ (Algorithm 1), for $K \geqslant 1$. Formally, we define $\mathcal{H}^K = \times_{k=0}^{K} \langle query^k_{Q_1}, \ldots, query^k_{Q_m}\rangle$, where:

- for each $k = 0, \ldots, K$ and each control state $q \in n\mathcal{F}(\mathcal{P})$, we have a procedure $query^k_Q$;
- in particular, $query^0_{Q_1}, \ldots, query^0_{Q_m}$ consists of one **assume** *false* statement;
- each procedure $query^k_Q$ has five sets of local variables, all of the same cardinality as $\mathbf{x}$: two sets, named $\mathbf{x}_I$ and $\mathbf{x}_O$, are used as input variables, whereas the other three sets, named $\mathbf{x}_J, \mathbf{x}_K$ and $\mathbf{x}_L$ are used locally by $query^k_Q$. Besides, $query^k_Q$ has a local variable called PC. There are no output variables.

Observe that each procedure $query^k_Q$ calls only procedures $query^{k-1}_{Q'}$ for some Q', hence the program $\mathcal{H}^K$ is non-recursive, and therefore amenable to summarization techniques that cannot handle recursion. Also the hierarchical structure of $\mathcal{H}^K$ enables modular summarization by computing the summaries ordered by increasing values of $k = 0, 1, \ldots, K$. The summaries of $\mathcal{H}^{K-1}$ are reused to compute $\mathcal{H}^K$. Finally, it is routine to check that the size of $\mathcal{H}^K$ is in $O(K \cdot \sharp\mathcal{P})$. Furthermore, the time needed to generate $\mathcal{H}^K$ is linear in the product $K \cdot \sharp\mathcal{P}$.

Given that $query^k_Q$ has two copies of $\mathbf{x}$ as input variables, and no output variables, the input output semantics $[\![\mathcal{H}]\!]^{i/o}_{query^k_Q} \subseteq \mathbb{Z}^{\mathbf{x}\times\mathbf{x}}$ is a set of tuples, rather than a (binary) relation. We denote $pre(query^k_Q) = \{I \cdot O \in \mathbb{Z}^{\mathbf{x}\times\mathbf{x}} \mid query^k_Q(I,O)$ returns with empty stack$\}$. Clearly $pre(query^k_Q) = [\![\mathcal{H}]\!]^{i/o}_{query^k_Q}$.

Algorithm 1. **proc** $query^k_Q(\mathbf{x}_I, \mathbf{x}_O)$ for $k \geqslant 1$

```
begin
            var $PC, \mathbf{x}_J, \mathbf{x}_K, \mathbf{x}_L$;
            $PC \leftarrow Q$;
start:      goto $\mathbf{p}^a_1$ or $\cdots$ or $\mathbf{p}^a_{n_a}$ or $\mathbf{p}^b_1$ or $\cdots$ or $\mathbf{p}^b_{n_b}$ or $\mathbf{p}^c_1$ or $\cdots$ or $\mathbf{p}^c_{n_c}$ ;
$\mathbf{p}^a_1$:   assume $(PC = head(p^a_1))$; assume $\rho_{tail(p^a_1)}(\mathbf{x}_I, \mathbf{x}_O)$; return;
            ⋮
$\mathbf{p}^a_{n_a}$:   assume $(PC = head(p^a_{n_a}))$; assume $\rho_{tail(p^a_{n_a})}(\mathbf{x}_I, \mathbf{x}_O)$; return;
$\mathbf{p}^b_1$:   assume $(PC = head(p^b_1))$; [ paste code for 2nd case: $tail(p^b_1) \in \Theta \times \mathcal{X}$ ];
            ⋮
$\mathbf{p}^b_{n_b}$:   assume $(PC = head(p^b_{n_b}))$; [ paste code for 2nd case: $tail(p^b_{n_b}) \in \Theta \times \mathcal{X}$ ];
$\mathbf{p}^c_1$:   assume $(PC = head(p^c_1))$; [ paste code for 3rd case: $tail(p^c_1) \in \langle \Theta \times \mathcal{X} \times \Theta \rangle \times \mathcal{X}$ ];
            ⋮
$\mathbf{p}^c_{n_c}$:   assume $(PC = head(p^c_{n_c}))$; [ paste code for 3rd case: $tail(p^c_{n_c}) \in \langle \Theta \times \mathcal{X} \times \Theta \rangle \times \mathcal{X}$ ];
end
```

2nd case. $tail(p^b_i) = \tau Q' \in \Theta \times \mathcal{X}$

```
havoc $(\mathbf{x}_J)$;
assume $\rho_\tau(\mathbf{x}_I, \mathbf{x}_J)$;
$\mathbf{x}_I \leftarrow \mathbf{x}_J$;
$PC \leftarrow Q'$ ;      // $query^k_{Q'}(\mathbf{x}_I, \mathbf{x}_O)$
goto start;              // return
```

3rd case. $tail(p^c_i) = \langle \tau\, Q^{init}_j\, \tau \rangle\, Q' \in \langle \Theta \times \mathcal{X} \times \Theta \rangle \times \mathcal{X}$

```
      havoc $(\mathbf{x}_J, \mathbf{x}_K, \mathbf{x}_L)$;
      assume $\rho_{\langle\tau}(\mathbf{x}_I, \mathbf{x}_J)$ ;        /* call relation */
      assume $\rho_{\tau\rangle}(\mathbf{x}_K, \mathbf{x}_L)$ ;        /* return relation */
      assume $\phi_\tau(\mathbf{x}_I, \mathbf{x}_L)$ ;        /* frame relation */
      goto ord or rod;
ord:  $query^{k-1}_{Q^{init}_j}(\mathbf{x}_J, \mathbf{x}_K)$;        /* in order exec. */
      $\mathbf{x}_I \leftarrow \mathbf{x}_L$ ;
      $PC \leftarrow Q'$ ;                       // $query^k_{Q'}(\mathbf{x}_I, \mathbf{x}_O)$
      goto start;                        // return
rod:  $query^{k-1}_{Q'}(\mathbf{x}_L, \mathbf{x}_O)$;        /* out of order exec. */
      $\mathbf{x}_I \leftarrow \mathbf{x}_J$ ;
      $\mathbf{x}_O \leftarrow \mathbf{x}_K$ ;
      $PC \leftarrow Q^{init}_j$ ;                 // $query^k_{Q^{init}_j}(\mathbf{x}_I, \mathbf{x}_O)$
      goto start;                        // return
```

In Alg. 1, p^α_i where $\alpha \in \{a,b,c\}$ refers to a production of the visibly pushdown grammar $G_{\mathcal{P}}$. The same symbol in boldface refers to the labelled statements in Alg. 1. The superscript $\alpha \in \{a,b,c\}$ differentiate the productions whether they are of the form $Q \to \tau$, $Q \to \tau Q'$ or $Q \to \langle \tau Q^{init}_j \tau \rangle Q'$, respectively.

Theorem 1 relates the semantics of $\mathcal{H}^K$ and the K-index semantics of $\mathcal{P}$. Given k, $1 \leqslant k \leqslant K$ and a control state q of $\mathcal{P}$, we show equality between $[\![\mathcal{H}^K]\!]^{i/o}_{query^k_Q}$ and $[\![\mathcal{P}]\!]^{(k)}_q$ over common variables. Before starting, we fix an arbitrary value for K and require that each k is such that $1 \leqslant k \leqslant K$. Hence, we drop K in $\mathcal{H}^K$ and write $\mathcal{H}$.

Inspection of the code of $\mathcal{H}$ reveals that $\mathcal{H}$ simulates k-index depth first derivations of $G_{\mathcal{P}}$ and interprets the statements of $\mathcal{P}$ on its local variables while applying derivation steps. The main difference with the normal execution of $\mathcal{P}$ is that $\mathcal{H}$ may interpret a procedure call and its continuation in an order which differs from the expected one.

Example 5. Let us consider an execution of *query* for the call $query^2_{Q_1^{init}}((1\ 0),(1\ 2))$ following $Q_1^{init} \xrightarrow{p_1^b p_2^b p_3^c} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle Q_4 \xrightarrow{p_4^a} \tau_1\tau_2\langle\tau_3 Q_1^{init}\tau_3\rangle\tau_4 \xrightarrow{p_1^b p_5^b p_6^b p_7^a} \tau_1\tau_2\langle\tau_3\tau_1\tau_5\tau_6\tau_7\tau_3\rangle\tau_4$. In the table below, the first row (labelled k/PC) gives the caller ($1 = query^1_{Q_4}$, $2 = query^2_{Q_1^{init}}$) and the value of PC when control hits the labelled statement given at the second row (labelled *ip*). The third row (labelled $\mathbf{x}_I/\mathbf{x}_O$) represents the content of the two arrays. $\mathbf{x}_I/\mathbf{x}_O = (a\ b)(c\ d)$ says that, in $\mathbf{x}_I$, x has value a and z has value b; in $\mathbf{x}_O$, x has value c and z has value d.

k/PC	$2/Q_1^{init}$	$2/Q_1^{init}$	$2/Q_2$	$2/Q_2$	$2/Q_3$	$2/Q_3$	$2/Q_3$
ip	**start**	$\mathbf{p_1^b}$	**start**	$\mathbf{p_2^b}$	**start**	$\mathbf{p_3^c}$	**rod**
$\mathbf{x}_I/\mathbf{x}_O$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$
k/PC	$1/Q_4$	$1/Q_4$	$2/Q_1^{init}$	$2/Q_1^{init}$	$2/Q_2$	$2/Q_2$	$2/Q_5$
ip	**start**	$\mathbf{p_4^a}$	**start**	$\mathbf{p_1^b}$	**start**	$\mathbf{p_5^b}$	**start**
$\mathbf{x}_I/\mathbf{x}_O$	$(1\ 0)(1\ 2)$	$(1\ 0)(1\ 2)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$
k/PC	$2/Q_5$	$2/Q_6$	$2/Q_6$				
ip	$\mathbf{p_6^b}$	**start**	$\mathbf{p_7^a}$				
$\mathbf{x}_I/\mathbf{x}_O$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$	$(0\ 0)(42\ 0)$				

The execution of $query^2_{Q_1^{init}}$ starts on row 1, column 1 and proceeds until the call to $query^1_{Q_4}$ at row 2, column 1 (the out of order case). The latter ends at row 2, column 2, where the execution of $query^2_{Q_1^{init}}$ resumes. Since the execution is out of order, and the previous $\mathbf{havoc}(\mathbf{x}_J,\mathbf{x}_K,\mathbf{x}_L)$ results into $\mathbf{x}_J = (0\ 0)$, $\mathbf{x}_K = (42\ 0)$ and $\mathbf{x}_L = (1\ 0)$ (this choice complies with the call relation), the values of $\mathbf{x}_I/\mathbf{x}_O$ are updated to $(0\ 0)/(42\ 0)$. The choice for equal values (0) of z in both $\mathbf{x}_I$ and $\mathbf{x}_0$ is checked in row 3, column 3. ■

Theorem 1. *Let $\mathcal{P} = \langle P_1,\ldots,P_n\rangle$ be a program, $\mathbf{x} = \bigcup_{i=1}^n \mathbf{x}_i$ be the set of all variables in $\mathcal{P}$, and let $q \in n\mathcal{F}(P_i)$ be a non-final control state of some procedure $P_i = \langle \mathbf{x}_i, \overrightarrow{\mathbf{x}}_i^{in}, \overrightarrow{\mathbf{x}}_i^{out}, S_i, q_i^{init}, F_i, \Delta_i\rangle$. Then, for any $k \geqslant 1$, we have:*

$$[\![\mathcal{H}]\!]^{i/o}_{query^k_Q} = \{I\cdot O \in \mathbb{Z}^{\mathbf{x}\times\mathbf{x}} \mid \langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q\}$$

Consequently, we also have:

$$[\![\mathcal{P}]\!]^{(k)}_q = \{\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \mid I\cdot O \in [\![\mathcal{H}]\!]^{i/o}_{query^k_Q}\}$$

The proof of Thm. 1 is based on the following lemma.

Lemma 2. *Let $k \geqslant 1$, q be a non-final control state of P_i and $I,O \in \mathbb{Z}^{\mathbf{x}}$. If $I\cdot O \in pre(query^k_Q)$ then $\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q$. Conversely, if $\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q$ then there exists $I',O' \in \mathbb{Z}^{\mathbf{x}}$ such that $I'\downarrow_{\mathbf{x}_i} = I\downarrow_{\mathbf{x}_i}$, $O'\downarrow_{\mathbf{x}_i} = O\downarrow_{\mathbf{x}_i}$ and $I'\cdot O' \in pre(query^k_Q)$.*

Proof: First we consider a tail-recursive version of Algorithm 1 which is obtained by replacing every two statements of the form $PC \leftarrow X$;**goto start**; by statements $query^k_X(\mathbf{x}_I,\mathbf{x}_O)$;**return**; (as it appears in the comments of Alg. 1). The equivalence between Algorithm 1 and its tail-recursive variant is an easy exercise.

"$\Leftarrow$" Let $\langle I\downarrow_{\mathbf{x_i}}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_q$. By definition of k-index semantics, there exists a tagged word $\alpha \in L^{(k)}_Q(G_{\mathcal{P}})$ such that $\langle I\downarrow_{\mathbf{x}_i}, O\downarrow_{\mathbf{x}_i}\rangle \in [\![\alpha]\!]$. Let p_1 be the first production used in the derivation of α and let $\ell \geqslant 1$ be the length (in number of productions used) of the derivation. Our proof proceeds by induction on ℓ. If $\ell = 1$ then we find that p_1 must be of the form $Q \to \tau$ and that $\alpha = \tau$. Therefore we have $[\![\alpha]\!] = [\![\tau]\!] = \rho_\tau$ and moreover $\langle I\!\downarrow_{\mathbf{x}_i}, O\!\downarrow_{\mathbf{x}_i}\rangle \in \rho_\tau$. Since $k \geqslant 1$, we let $I' = I$, $O' = O$ and we find that $query^k_Q(I', O')$ returns by choosing to jump to the label corresponding to p_1, then executing the **assume** statement and finally the **return** statement. Thus $I' \cdot O' \in pre(query^k_Q)$. For $\ell > 1$, the proof divides in two parts.

1. If p_1 is of the form $Q \to \tau Q'$ then we find that $\alpha = \tau\beta$, for some tagged word β. Moreover, $\langle I\!\downarrow_{\mathbf{x}_i}, O\!\downarrow_{\mathbf{x}_i}\rangle \in [\![\alpha]\!] = \rho_\tau \circ [\![\beta]\!]$ by definition of the semantics. This implies that there exists $J \in \mathbb{Z}^{\mathbf{x}}$ such that $\langle I\!\downarrow_{\mathbf{x}_i}, J\!\downarrow_{\mathbf{x}_i}\rangle \in \rho_\tau$ and $\langle J\!\downarrow_{\mathbf{x}_i}, O\!\downarrow_{\mathbf{x}_i}\rangle \in [\![\beta]\!]$. Hence, we conclude from $\beta \in L^{(k)}_{Q'}(G_{\mathcal{P}})$, and the fact that the derivation $Q' \overset{(k)}{\Longrightarrow}{}^\star \beta$ has less productions than $Q \overset{(k)}{\Longrightarrow}{}^\star \alpha$, that $\langle J\!\downarrow_{\mathbf{x}_i}, O\!\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_{Q'}$. Applying the induction hypothesis on this last fact, we find that $J \cdot O \in pre(query^k_{Q'})$. Finally consider the call $query^k_Q(I, O)$ where at label **start** the jump goes to label corresponding to p_1. At this point in the execution **havoc**($\mathbf{x}_J$) returns J. Next **assume** $\rho_\tau(I, J)$ succeeds. Finally we find that the call to $query^k_Q(I, O)$ returns because so does the call $query^k_{Q'}(J, O)$ which is followed by **return**. Hence $I \cdot O \in pre(query^k_Q)$.

2. If p_1 is of the form $Q \to \langle\tau Q^{init}_j \tau\rangle Q'$ then we find that $\alpha = \langle\tau \beta' \tau\rangle \beta$ for some β', β. Lemma 1 (prop. 3) shows that either $\beta' \in L^{(k-1)}_{Q^{init}_j}(G_{\mathcal{P}})$ and $\beta \in L^{(k)}_{Q'}(G_{\mathcal{P}})$ or $\beta' \in L^{(k)}_{Q^{init}_j}(G_{\mathcal{P}})$ and $\beta \in L^{(k-1)}_{Q'}(G_{\mathcal{P}})$, and both derivations have less productions than ℓ. We will assume the former case, the latter being treated similarly. Moreover, $\langle I\!\downarrow_{\mathbf{x}_i}, O\!\downarrow_{\mathbf{x}_i}\rangle \in [\![\alpha]\!] = CaRet_\tau \circ [\![\beta]\!] = \Big(\big(\rho_{\langle\tau} \circ [\![\beta']\!] \circ \rho_{\tau\rangle}\big) \cap \phi_\tau\Big) \circ [\![\beta]\!] \subseteq \Big(\big(\rho_{\langle\tau} \circ [\![\mathcal{P}]\!]^{(k-1)}_{q^{init}_j} \circ \rho_{\tau\rangle}\big) \cap \phi_\tau\Big) \circ [\![\mathcal{P}]\!]^{(k)}_{q'}$. Hence there exists $J, K, L \in \mathbb{Z}^{\mathbf{x}}$ such that $\langle I\!\downarrow_{\mathbf{x}_i}, J\!\downarrow_{\mathbf{x}_i}\rangle \in \rho_{\langle\tau}$, $\langle J\!\downarrow_{\mathbf{x}_i}, K\!\downarrow_{\mathbf{x}_j}\rangle \in [\![\mathcal{P}]\!]^{(k-1)}_{q^{init}_j}$, $\langle K\!\downarrow_{\mathbf{x}_j}, L\!\downarrow_{\mathbf{x}_j}\rangle \in \rho_{\tau\rangle}$, and $\langle L\!\downarrow_{\mathbf{x}_i}, O\!\downarrow_{\mathbf{x}_i}\rangle \in [\![\mathcal{P}]\!]^{(k)}_{q'}$. Applying the induction hypothesis on the derivations of β' and β, we obtain that $J \cdot K \in pre(query^{k-1}_{Q^{init}_j})$ and $L \cdot O \in pre(query^k_{Q'})$. Given those facts, it is routine to check that $query^k_Q(I', O')$ returns by choosing to jump to the label corresponding to p_1, then having **havoc**($\mathbf{x}_J, \mathbf{x}_K, \mathbf{x}_L$) return (J, K, L), hence $I' \cdot O' \in pre(query^k_Q)$.

The only if direction is proven in the technical report [11]. □

As a last point, we prove that the bounded-index sequence $\{[\![\mathcal{P}]\!]^{(k)}\}_{k=1}^{\infty}$ satisfies several conditions that advocate its use in program analysis, as an underapproximation sequence. The subset order and set union is extended to tuples of relations, point-wise.

$$\begin{array}{lll} [\![\mathcal{P}]\!]^{(k)} \subseteq [\![\mathcal{P}]\!]^{(k+1)} & \text{for all } k \geqslant 1 & (A1) \\ [\![\mathcal{P}]\!] \;\; = \bigcup_{k=1}^{\infty} [\![\mathcal{P}]\!]^{(k)} & & (A2) \end{array}$$

Condition ($A1$) requires that the sequence is monotonically increasing, the limit of this increasing sequence being the actual semantics of the program ($A2$). These conditions follow however immediately from the two first points of Lemma 1. To decide whether the limit $[\![\mathcal{P}]\!]$ has been reached by some iterate $[\![\mathcal{P}]\!]^{(k)}$, it is enough to check that the tuple of relations in $[\![\mathcal{P}]\!]^{(k)}$ is inductive with respect to the statements of $\mathcal{P}$. This can be implemented as an SMT query.

5 Completeness of Underapproximations for Bounded Programs

In this section we define a class of recursive programs for which the precise summary semantics of each program in that class is effectively computable. We show for each program $\mathcal{P}$ in the class that (a) $[\![\mathcal{P}]\!] = [\![\mathcal{P}]\!]^{(k)}$ for some value $k \geqslant 1$, bounded by a linear function in the total number $loc(\mathcal{P})$ of control states in $\mathcal{P}$, and moreover (b) the semantics of $\mathcal{H}^k$ is effectively computable (and so is that of $[\![\mathcal{P}]\!]^{(k)}$ by Thm. 1).

Given an integer relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$, its *transitive closure* $R^+ = \bigcup_{i=1}^{\infty} R^i$, where $R^1 = R$ and $R^{i+1} = R^i \circ R$, for all $i \geqslant 1$. In general, the transitive closure of a relation is not definable within decidable subsets of integer arithmetic, such as Presburger arithmetic. In this section we consider two classes of relations, called *periodic*, for which this is possible, namely octagonal relations, and finite monoid affine relations. The formal definitions are deferred to the technical report [11].

We define a *bounded-expression* $\mathbf{b}$ to be a regular expression of the form $\mathbf{b} = w_1^* \ldots w_k^*$, where $k \geqslant 1$ and each w_i is a non-empty word. A language (not necessarily context-free) L over alphabet Σ is said to be *bounded* if and only if L is included in (the language of) a bounded expression $\mathbf{b}$.

Theorem 2 ([18]). *Let $G = (\mathcal{X}, \Sigma, \delta)$ be a grammar, and $X \in \mathcal{X}$ be a nonterminal, such that $L_X(G)$ is bounded. Then there exists a linear function $\mathcal{B} : \mathbb{N} \to \mathbb{N}$ such that $L_X(G) = L_X^{(k)}(G)$ for some $1 \leqslant k \leqslant \mathcal{B}(\mathcal{X})$.*

If the grammar in question is $G_{\mathcal{P}}$, for a program $\mathcal{P}$, then clearly $\mathcal{X} = loc(\mathcal{P})$, by definition. The class of programs for which our method is complete is defined below:

Definition 2. *Let $\mathcal{P}$ be a program and $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$ be its corresponding visibly pushdown grammar. Then $\mathcal{P}$ is said to be* bounded periodic *if and only if:*
1. *$L_X(G_{\mathcal{P}})$ is bounded for each $X \in \mathcal{X}$;*
2. *each relation ρ_τ occurring in the program, for some $\tau \in \widehat{\Theta}$, is periodic.*

Example 6. (continued from Ex. 4) Recall that $L_{Q_1^{init}}(G_{\mathcal{P}}) = L_{Q_1^{init}}^{(2)}(G_{\mathcal{P}})$ which equals to the set $\{(\tau_1\tau_2\langle\tau_3)^n\tau_1\tau_5\tau_6\tau_7(\tau_3\rangle\tau_4)^n \mid n \geqslant 0\} \subseteq (\tau_1\tau_2\langle\tau_3)^*\tau_1^*\tau_5^*\tau_6^*\tau_7^*(\tau_3\rangle\tau_4)^*$. ■

Concerning condition 1, it is decidable [12] and previous work [14] defined a class of programs following a recursion scheme which ensures boundedness of the set of interprocedurally valid paths.

This section shows that the underapproximation sequence $\{[\![\mathcal{P}]\!]^{(k)}\}_{k=1}^{\infty}$, defined in Section 4, when applied to any bounded periodic programs $\mathcal{P}$, always yields $[\![\mathcal{P}]\!]$ in at most $\mathcal{B}(loc(\mathcal{P}))$ steps, and moreover each iterate $[\![\mathcal{P}]\!]^{(k)}$ is computable and Presburger definable. Furthermore the method can be applied *as it is* to bounded periodic programs, without prior knowledge of the bounded expression $\mathbf{b} \supseteq L_Q(G_{\mathcal{P}})$.

The proof goes as follows. Because $\mathcal{P}$ is bounded periodic, Thm. 2 shows that the semantics $[\![\mathcal{P}]\!]$ of $\mathcal{P}$ coincide with its k-index semantics $[\![\mathcal{P}]\!]^{(k)}$ for some $1 \leqslant k \leqslant$

$\mathcal{B}(loc(\mathcal{P}))$. Hence, the result of Thm. 1 shows that for each $q \in n\mathcal{F}(\mathcal{P})$, the k-index semantics $[\![\mathcal{P}]\!]_q^{(k)}$ is given by the semantics $[\![\mathcal{H}]\!]_{query_Q^k}$ of procedure $query_Q^k$ of the program $\mathcal{H}$. Then, because $\mathcal{P}$ is bounded, we show in Thm. 3 that every procedure $query_Q^k$ of program $\mathcal{H}$ is *flattable* (Def. 3). Moreover, since the only transitions of $\mathcal{H}$ which are not from $\mathcal{P}$ are equalities and **havoc**, all transitions of $\mathcal{H}$ are periodic. Since each procedure $query_Q^k$ is flattable then $[\![\mathcal{P}]\!]$ is computable in finite time by existing tools, such as FAST [6] or FLATA [8, 7]. In fact, these tools are guaranteed to terminate provided that (a) the input program is flattable; and (b) loops are labeled with periodic relations.

Definition 3. *Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a non-recursive program and $G_{\mathcal{P}} = (\mathcal{X}, \widehat{\Theta}, \delta)$ be its corresponding visibly pushdown grammar. Procedure P_i is said to be* flattable *if and only if there exists a bounded and regular language R over $\widehat{\Theta}$, such that $[\![\mathcal{P}]\!]_{P_i} = \bigcup_{\alpha \in L_{P_i}(G_{\mathcal{P}}) \cap R} [\![\alpha]\!]$.*

Notice that a flattable program is not necessarily bounded (Def. 2), but its semantics can be computed by looking only at a bounded subset of interprocedurally valid sequence of statements.

Theorem 3. *Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a bounded program, and let $q \in n\mathcal{F}(\mathcal{P})$. Then, for any $k \geqslant 1$, procedure $query_Q^k$ of program $\mathcal{H}$ is and flattable.*

The proof of Thm. 3 roughly goes as follows: recall that we have $[\![\mathcal{P}]\!]_q = [\![\mathcal{P}]\!]_q^{(k)}$ for each $q \in n\mathcal{F}(\mathcal{P})$ and so it is sufficient to consider the set $L_Q^{(k)}(G_{\mathcal{P}})$ of interprocedurally valid paths. We further show (Thm. 4) that a strict subset of the k-index derivations of $G_{\mathcal{P}}$ is sufficient to capture $L_Q^{(k)}(G_{\mathcal{P}})$. Moreover this subset of derivations is characterizable by a regular bounded expression $\mathbf{b}_\Gamma$ over the productions of $G_{\mathcal{P}}$. Next, we map $\mathbf{b}_\Gamma$ into a set $f(\mathbf{b}_\Gamma)$ of interprocedurally valid paths of procedure $query_Q^k$ of $\mathcal{H}$, which is sufficient to capture $[\![\mathcal{H}]\!]_{query_Q^k}$. Finally, using existing results [12], we show in Thm. 5 that $f(\mathbf{b}_\Gamma)$ is a bounded and regular set. Hence, we conclude that each procedure $query_Q^k$ is flattable. A full proof of Thm. 3 is given in the technical report [11].

Given a grammar $G = (\mathcal{X}, \Sigma, \delta)$, we call any subset of δ^* a *control set*. Let Γ be a control set, we denote by $L_X(\Gamma, G) = \{w \in \Sigma^* \mid \exists \gamma \in \Gamma \colon X \overset{\gamma}{\Longrightarrow} w\}$, the set of words resulting from derivations with control word in Γ.

Definition 4 ([19]). *Let $D \equiv X = w_0 \Longrightarrow^* w_m = w$ be a derivation. Let $k > 0$, $x_i \in \Sigma^*$, $A_i \in \mathcal{X}$ such that $w_m = x_0 A_1 x_1 \cdots A_k x_k$; and for each m and i such that $0 \leqslant m \leqslant n$, $1 \leqslant i \leqslant k$, let $f_m(i)$ denote the index of the first word in D in which the particular occurrence of variable A_i appears. Let A_j be the nonterminal replaced in step $w_m \Longrightarrow w_{m+1}$ of D. Then D is said to be* depth-first *if and only if for all m, $0 \leqslant m < n$ we have $f_m(i) \leqslant f_m(j)$, for all $1 \leqslant i \leqslant k$.*

We define the set $DF_X(G)$ ($DF_X^{(k)}(G)$) of words produced using only depth-first derivations (of index at most k) in G starting from X. Clearly, $DF_X(G) \subseteq L_X(G)$ and similarly $DF_X^{(k)}(G) \subseteq L_X^{(k)}(G)$ for all $k \geqslant 1$. We further define the set $DF_X(\Gamma, G)$ ($DF_X^{(k)}(\Gamma, G)$) of words produced using depth-first derivations (of index at most k) with control words from Γ.

The following theorem shows that $L_Q^{(k)}(G_{\mathcal{P}})$ is captured by a subset of depth-first derivations whose control words belong to some bounded expression.

Theorem 4. *Let $G = (\mathcal{X}, \Theta, \delta)$ be a visibly pushdown grammar, $X_0 \in \mathcal{X}$ be a nonterminal such that $L_{X_0}(G)$ is bounded. Then for each $k \geqslant 1$ there exists a bounded expression $\mathbf{b}_\Gamma$ over δ such that $DF_{X_0}^{(k)}(\mathbf{b}_\Gamma, G) = L_{X_0}^{(k)}(G)$.*

Finally, to conclude that $query_Q^k$ is flattable, we map the k-index depth-first derivations of G into the interprocedurally valid paths of $query_Q^k$. Then, applying Thm. 5 on that mapping, we conclude the existence of a bounded and regular set of interprocedurally valid paths of $query_Q^k$ sufficient to capture its semantics.

Theorem 5. *Given two alphabets Σ and Δ, let f be a function from Σ^* into Δ^* such that (i) if u is a prefix of v then $f(u)$ is a prefix of $f(v)$; (ii) there exists an integer M such that $|f(wa)| - |f(w)| \leqslant M$ for all $w \in \Sigma^*$ and $a \in \Sigma$; (iii) $f(\varepsilon) = \varepsilon$; (iv) $f^{-1}(R)$ is regular for all regular languages R. Then f preserves regular sets. Furthermore, for each bounded expression $\mathbf{b}$ we have that $f(\mathbf{b})$ is bounded.*

6 Experiments

We have implemented the proposed method in the FLATA verifier [16] and experimented with several benchmarks. First, we have considered several programs, taken from [1], that perform arithmetic and logical operations in a recursive way such as `plus` (addition), `timesTwo` (multiplication by two), `leq` (comparison), and `parity` (parity checking). It is worth noting that these programs have finite index and stabilization of the underapproximation sequence is thus guaranteed. Our technique computes summaries by verifying that $[\![\mathcal{P}]\!]^{(2)} = [\![\mathcal{P}]\!]^{(3)}$ for all these benchmarks, see Table 1 (the platform used for experiments is Intel® Core™ i7-3770K CPU, 3.50GHz, with 16GB of RAM).

Table 1. Experiments

Program	Time [s]	k
`timesTwo`	0.7	2
`leq`	0.8	2
`parity`	0.8	2
`plus`	1.5	2
$F_{a=2}$	1.5	3
$F_{a=8}$	36.9	4
$G_{b=12}$	3.5	3
$G_{b=13}$	23.2	3
$G_{b=14}$	23.4	3

$$F_a(x) = \begin{cases} x - 10 & \text{if } x \geqslant 101 \\ (F_a)^a(x + 10 \cdot a - 9) & \text{if } x \leqslant 100 \end{cases} \qquad G_b(x) = \begin{cases} x - 10 & \text{if } x \geqslant 101 \\ G(G(x + b)) & \text{if } x \leqslant 100 \end{cases}$$

Next, we have considered the generalized McCarthy 91 function [9], a well-known verification benchmark that has long been a challenge. We have automatically computed precise summaries of its generalizations F_a and G_b above for $a = 2, \ldots, 8$ and $b = 12, \ldots, 14$. The indices of the recursive programs implementing the F_a, G_b functions are not bounded, however the sequence reached the fixpoint after at most 4 steps.

7 Conclusions

We have presented an underapproximation method for computing summaries of recursive programs operating on integers. The underapproximation is driven by bounding the index of derivations that produce the execution traces of the program, and computing the summary, for each index, by analyzing a non-recursive program. We also present a class of programs on which our method is complete. Finally, we report on an implementation and experimental evaluation of our technique.

References

1. Termination Competition 2011, `http://termcomp.uibk.ac.at/termcomp/home.seam`
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. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56(3), 16 (2009)
4. Atig, M.F., Ganty, P.: Approximating petri net reachability along context-free traces. In: FSTTCS 2011. LIPIcs, vol. 13, pp. 152–163. Schloss Dagstuhl (2011)
5. Cook, B., Podelski, A., Rybalchenko, A.: Summarization for termination: no return! Formal Methods in System Design 35, 369–387 (2009)
6. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Fast Acceleration of Symbolic Transition Systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 118–121. Springer, Heidelberg (2003)
7. Bozga, M., Iosif, R., Konečný, F.: Fast Acceleration of Ultimately Periodic Relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)
8. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. Fundamenta Informaticae 91(2), 275–303 (2009)
9. Cowles, J.: Knuth's generalization of McCarthy's 91 function. In: Computer-aided Reasoning, pp. 283–299 (2000)
10. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian program analysis. JACM 57(6), 33:1–33:47 (2010)
11. Ganty, P., Iosif, R., Konečný, F.: Underapproximation of procedure summaries for integer programs. CoRR abs/1210.4289 (2012)
12. Ginsburg, S.: The Mathematical Theory of Context-Free Languages. McGraw-Hill, Inc., New York (1966)
13. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL 2010, pp. 43–56. ACM (2010)
14. Godoy, G., Tiwari, A.: Invariant Checking for Programs with Procedure Calls. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 326–342. Springer, Heidelberg (2009)
15. Gruska, J.: A few remarks on the index of context-free grammars and languages. Information and Control 19(3), 216–223 (1971)
16. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A Verification Toolkit for Numerical Transition Systems - Tool Paper. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 247–251. Springer, Heidelberg (2012)
17. Lalire, G., Argoud, M., Jeannet, B.: Interproc., `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`
18. Luker, M.: A family of languages having only finite-index grammars. Information and Control 39(1), 14–18 (1978)
19. Luker, M.: Control sets on grammars using depth-first derivations. Mathematical Systems Theory 13, 349–359 (1980)
20. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995, pp. 49–61. ACM (1995)
21. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, ch. 7, pp. 189–233. Prentice-Hall, Inc. (1981)

Runtime Verification Based on Register Automata

Radu Grigore[1], Dino Distefano[1], Rasmus Lerchedahl Petersen[2], and Nikos Tzevelekos[1]

[1] Queen Mary University of London
[2] Microsoft Research

Abstract. We propose TOPL automata as a new method for runtime verification of systems with unbounded resource generation. Paradigmatic such systems are object-oriented programs which can dynamically generate an unbounded number of fresh object identities during their execution. Our formalism is based on register automata, a particularly successful approach in automata over infinite alphabets which administers a finite-state machine with boundedly many input-storing registers. We show that TOPL automata are equally expressive to register automata and yet suitable to express properties of programs. Compared to other runtime verification methods, our technique can handle a class of properties beyond the reach of current tools. We show in particular that properties which require value updates are not expressible with current techniques yet are naturally captured by TOPL machines. On the practical side, we present a tool for runtime verification of Java programs via TOPL properties, where the trade-off between the coverage and the overhead of the monitoring system is tunable by means of a number of parameters. We validate our technique by checking properties involving multiple objects and chaining of values on large open source projects.

1 Introduction

Runtime verification [19,22] connotes the monitoring of program executions in order to detect specific error traces which correspond to violations of sought safety properties. In contrast to its static counterpart, runtime verification checks only certain program executions, yet the error reports are accurate as detected violations represent real bugs in the program. In the case of systems with dynamic generation of resources, such as object references in Java, runtime verification faces the key challenge of reasoning about a potentially *unbounded* number of parameter values representing resource identities. Hence, the techniques applicable in this realm of programs must be able to deal with infinite alphabets (this idiom is also known as parametric monitoring). Leading runtime verification techniques tackle the issue using different approaches, such as reducing the problem to checking projections of execution traces over bounded sets of data values (*trace slicing*) [26,20,3,2], or employing abstract machines whose transition rules are explicitly parameterised [9,7,8].

Another community particularly interested in reasoning over similar data domains, albeit motivated by XML reasoning and model-checking, is the one working on automata over infinite alphabets. Their research has been prolific in developing a wide range of paradigms and accompanying logics, with varying degrees of expressivity and effectiveness (see [28] for an overview from 2006). A highly successful such paradigm is that of *Register Automata* [21,24], which are finite-state machines equipped with a fixed number of registers where input values can be stored, updated and compared with subsequent inputs. They provide a powerful device for reasoning about temporal relations between a possibly unbounded number of objects in a finite manner. In this work we propose a foundational runtime verification method based on a novel class of machines called *TOPL automata*, which connects the field with the literature on automata over infinite alphabets and, more specifically, with register automata.

The key features of our machines are: (1) the use of registers, and (2) the use of sets of active states (non-determinism). From the verification point of view, registers allow us to use a fixed amount of specification variables which, however, can be *re-bound* (i.e. have their values updated). On the other hand, by being able to spawn several active states, we can select different parts of the same run to be stored and processed. These features give us the expressive power to capture a wide range of realistic program properties in a *finite* way. A specific such class of properties concerns *chaining* or *propagation*, which are of focal importance in areas like dynamic taint analysis [25] as well as dynamic shape analysis.[1] In the latter case, we aspire to reason at runtime about particular shapes of dynamically allocated data-structures irrespectively of their size. For example, checking

"the shape of the list should not contain cycles" (1)

for lists of any size and in a finite way, requires two activities. First, being able to change the value of the variables in the specification while traversing the list (re-binding). Second, keeping correlations of different elements in the list at the same time (multiple active states).

The aim of this work is to exploit the flexibility and the power of registers to address certain properties not expressible with other approaches while, on the other hand, making it easy for programmers to express properties of their code. More precisely, we start from register automata and extend them driven by typical properties required in real-world object-oriented systems. This process results in the definition of two new classes of automata:

– *TOPL* automata, which are low-level and are used for simplifying the formal correspondence with register automata;
– *hl-TOPL* automata, which are high-level and naturally express temporal properties about programs.

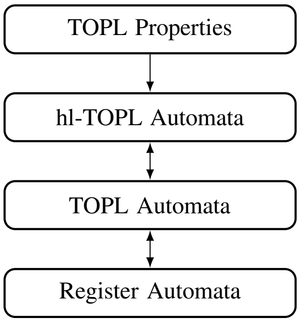


Fig. 1. Diagram of the main concepts. The target of each arrow is at least as expressive as its source.

We moreover define the ***Temporal Object Property Language (TOPL)***, a formal language which maps directly onto hl-TOPL automata and is used for expressing runtime specifications. TOPL is a Java-programmer-friendly language where properties look

[1] Although shape analysis is mainly a static technique, we will see in Section 2 that, when doing run-time monitoring, being able to reason about shapes may be vital.

like small Java programs that violate the desired program behaviour. The hierarchy of presented concepts is depicted in Figure 1.

We complement and validate our theoretical results with a practical runtime verification tool for Java programs. The tool can be used by programmers to rigorously express temporal properties about programs, which are then automatically checked by the system. Although the formal correspondence to register automata is completely hidden from the user, it provides a concrete automata-theoretic foundation which allows us to know formally the advantages and limitations of our technique, and also reuse results from the register automata literature. Moreover, our tool can be tuned in terms of coverage, overhead and trace reporting by means of a number of parameters.

Contributions. This paper builds upon [18], which introduced the language of TOPL properties and drafted the corresponding automata. Here we clarify the latter, provide a formal correspondence to automata over infinite alphabet, and devise and test a practical tool implementation. In summary, the contributions of the present work are:

– We introduce TOPL and hl-TOPL automata, two classes of abstract machines for verifying systems over infinite alphabets. We prove that both formalisms are equally expressive to register automata by constructing formal reductions between them. The reductions allow us to transfer results from the register automata setting to ours (e.g. decidability of language emptiness, language closures, etc.).

– We define TOPL, a formal specification language designed for expressing program properties involving object interactions over time in a way that is familiar to object-oriented programmers. We moreover present a formal semantics for TOPL, thus making it suitable for static and dynamic program analysis.

– We implement a tool for automatically checking for violations of TOPL properties in Java programs at runtime. A number of parameters are provided for tuning the precision of the system. We furthermore report on experiments in which we ran our tool on large open-source projects. The results are encouraging: for example, we have found an interesting and previously unknown concurrency bug in the DaCapo suite [13].

2 Motivating Examples

Interaction among objects is at the core of the object-oriented paradigm. Consider for example Java collections. A typical property one would want to state is

If one iterator modifies its collection then other iterators of the same collection become invalid, i.e. they cannot be used further. (2)

The formalisation of the above constraint is non-trivial since it needs to keep track of *several objects* (at least two iterators and one collection) and their *interaction over time*.

A slightly more complex scenario is described in Figure 2. Class `CharArray` manipulates an array of chars, while class `Concat` concatenates two objects of type `Str`. Both classes implement the `Str` interface. Consider the case where `Concat` is used for implementing a *rope*.[2] The operations of a rope (e.g. insert, concat, delete) may update its

[2] A rope is a data structure for efficiently storing and manipulating very long strings.

```
interface Str {
  void set(int i, char c);
  char get(int i);
  int len();
  Itr iterator();
}
interface Itr {
  boolean hasNext();
  char next();
  void set(char c);
}
```

```
class CharArray implements Str {
  char [] data;
  // ...
}
class Concat implements Str {
  Str s, t;
  public static Concat make(Str s, Str t) {
  /* ... */
  }
  // ...
}
```

Fig. 2. A first example: Java code

shape and the references to its root. In this case we may have two or more collections *sharing* some elements. Hence, iterators operating on those different collections may invalidate each other. We need to modify (2), increasing its complexity:

If one iterator modifies its collection then other iterators of collections sharing some of its elements become invalid, i.e. they cannot be used further. (3)

On the need for re-binding. Let us now suppose we want to perform *taint checking* on input coming from a web form. What we want to check is the property:

Any value introduced by the `input()` *method should not reach the* `sink()` *method without first passing through the* `sanitizer()` *method.* (4)

Although the property may seem simple, its difficulty can vary depending on the context. Consider the case where the input is constructed by concatenating strings from a web form, for example by using ropes implemented with class `Concat`. The number of user inputs, and therefore of concatenations, is not known a priori and is in general unbounded. Consequently, we may end up having an unbounded number of tainted objects. In a temporal specification, we would then need either one logical variable for each of them, or the ability to *rebind* (or *update*) variables in the specification so that we can trace taint propagation. For an unbounded number of objects, rebinding specification variables with different values during the computation helps in keeping the specification finite.

The need for rebinding of variables in the specification arises also in other contexts. For example, when reasoning about the evolving shape of dynamically allocated data-structures. Consider the following loop which uses a list:

```
while (l.next()!=null) { ... }
```

If the list `l` contains a cycle, the loop will diverge. Being a violation of a liveness property (termination), divergence cannot be observed at runtime in finite time and therefore it is harder to debug. If we obtained the list by calling a third-party library, we would want to check the property (1) from the Introduction. The *finite* encoding of such properties requires the ability to update the values of specification variables.

3 TOPL Automata

We start by presenting some basic definitions. We fix V to be an infinite set of *values*, with its members denoted by v, u and variants. Given an arity n, a *letter* ℓ is an element $(v_1, \dots, v_n) \in \Sigma$, where $\Sigma = V^n$ is the *alphabet*. For $\ell = (v_1, ..., v_n)$, we set the notation $\ell(i) = v_i$. Given a size m, we define the set of stores to be $S = V^m$. For a store $s = (u_1, ..., u_m)$, we write $s(i)$ for u_i. A *register* i is an integer from the set $\{1, ..., m\}$ identifying a component of the store.

A *guard* g is a formula in a specified logic, interpreted over pairs of letters and stores; we write $(s, \ell) \models g$ to denote that the store $s \in S$ and the letter $\ell \in \Sigma$ satisfy the guard g, and we denote the set of guards by G. An *action* a is a small program which, given an input letter, performs a store update. That is, the set of actions is some set $A \subseteq \Sigma \to S \to S$.

Given an alphabet $\Sigma = V^n$ and a (memory) size m, we shall define TOPL automata to operate on the set of labels $\Lambda = G \times A$, where G and A are given by:

$$\begin{aligned} G &::= \mathsf{eq}\, i\, j \mid \mathsf{neq}\, i\, j \mid \mathsf{true} \mid G \mathbin{\mathsf{and}} G \\ A &::= \mathsf{nop} \mid \mathsf{set}\, i := j \mid A; A \end{aligned}$$

with $i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$. If $n = 1$, then $(\mathsf{eq}\, i)$ stands for $(\mathsf{eq}\, i\, 1)$; $(\mathsf{neq}\, i)$ for $(\mathsf{neq}\, i\, 1)$; and $(\mathsf{set}\, i)$ stands for $(\mathsf{set}\, i := 1)$. The guards are evaluated as follows.

$$\begin{array}{llll} (s,\ell) \models \mathsf{eq}\, i\, j & \text{if } s(i) = \ell(j), & (s,\ell) \models \mathsf{true} & \text{always,} \\ (s,\ell) \models \mathsf{neq}\, i\, j & \text{if } s(i) \neq \ell(j), & (s,\ell) \models g_1 \mathsf{and}\, g_2 & \text{if } (s,\ell) \models g_1 \text{ and } (s,\ell) \models g_2. \end{array}$$

The TOPL actions are built up from the empty action, $\mathsf{nop}(\ell)(s) = s$; the assignment action, $(\mathsf{set}\, i := j)(\ell)(s) = s[i \mapsto \ell(j)]$ (where $s[i \mapsto v](k) = s(k)$ if $k \neq i$, and v otherwise); and action composition, $(a_1; a_2)(\ell) = a_1(\ell) \circ a_2(\ell)$.

We can now define our first class of automata.

Definition 1. *A **TOPL automaton** with m registers, operating on n-tuples, is a tuple $\mathcal{A} = \langle Q, q_0, s_0, \delta, F \rangle$ where:*
- *Q is a finite set of states, with initial one $q_0 \in Q$ and final ones $F \subseteq Q$;*
- *$s_0 \in S$ is an initial store;*
- *$\delta \subseteq Q \times \Lambda \times Q$ is a finite transition relation.*

A ***configuration*** x is a pair (q, s) of a state q and a store s; we denote the set of configurations by $X = Q \times S$. The *initial* configuration is (q_0, s_0). A configuration is *final* when its state is final. The configuration graph of a TOPL automaton $\mathcal{A}$ as above is a subset of $X \times \Sigma \times X$. We write $x_1 \xrightarrow{\ell}_{\mathcal{A}} x_2$ to mean that (x_1, ℓ, x_2) is in the configuration graph of $\mathcal{A}$ (we may omit the subscript if $\mathcal{A}$ is clear from the context).

Definition 2. *Let $\mathcal{A}$ be a TOPL automaton. The **configuration graph** of $\mathcal{A}$ consists of exactly those configuration transitions $(q_1, s_1) \xrightarrow{\ell}_{\mathcal{A}} (q_2, s_2)$ for which there is a TOPL-automaton transition $(q_1, (g, a), q_2) \in \delta$ such that $(s_1, \ell) \models g$ and $a(\ell)(s_1) = s_2$.*

*The **language** $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of words that label walks from the initial configuration to some final one: $\mathcal{L}(\mathcal{A}) = \{\, \ell_1 \dots \ell_k \mid x_0$ initial, x_k final, $\forall i \leq k.\, x_{i-1} \xrightarrow{\ell_i}_{\mathcal{A}} x_i \,\}$.*

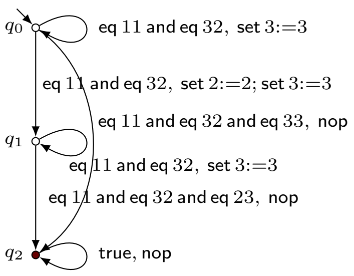
A TOPL automaton is ***deterministic*** when its configuration graph contains no two distinct transitions that have the same source x_1 and are labeled by the same letter ℓ, that is, $x_1 \xrightarrow{\ell} x_2$ and $x_1 \xrightarrow{\ell} x_3$ with $x_2 \neq x_3$.

Example 3. Consider the language $\{\, abc \in V^3 \mid a \neq c \text{ and } b \neq c \,\}$. It is recognized by the following TOPL automaton with 2 registers over the alphabet $\Sigma = V$. The values in the initial store s_0 can be arbitrary.

– $Q = \{1, 2, 3, 4\}$, $q_0 = 1$ and $F = \{4\}$;
– $\delta = \{(1, (\mathsf{true}, \mathsf{set}\,1), 2)\} \cup \{(2, (\mathsf{true}, \mathsf{set}\,2), 3)\} \cup \{(3, (\mathsf{neq}\,1 \text{ and } \mathsf{neq}\,2, \mathsf{nop}), 4)\}$.

Example 4. An involved example is one accepting the language

$$\mathcal{L}_{v_0} = \{\, (\mathsf{next}, v_0, v_1)(\mathsf{next}, v_1, v_2) \cdots (\mathsf{next}, v_n, v) \mid \forall i \neq j.\, v_i \neq v_j \ \wedge\ \exists k.\, v = v_k \,\}$$


q_0
eq 11 and eq 32, set 3:=3
eq 11 and eq 32, set 2:=2; set 3:=3
eq 11 and eq 32 and eq 33, nop
q_1
eq 11 and eq 32, set 3:=3
eq 11 and eq 32 and eq 23, nop
q_2
true, nop

which detects whether a linked list starting from v_0 contains a cycle. Here, $\mathsf{next}, v_0, \ldots, v_n \in V$, and next, v_0 are fixed. The automaton, depicted on the side, contains three registers and its initial store is $(\mathsf{next}, v_0, v_0)$. Its initial state is q_0, and its final one q_2. Register 1 always contains the constant next, and eq 11 checks at every transition whether the first value of the letter is `next`. The third register is used for storing the value of the current next node. Finally, the second register is non-deterministically fed with the value of a list node ($\mathsf{set}\,2 := 2$ in $q_0 \to q_1$), to be matched with a subsequent value ($\mathsf{eq}\,23$ in $q_1 \to q_2$) and thus lead to an error (i.e. final state q_2). We can also reach an error by receiving a letter with its next value being the same as its own value ($\mathsf{eq}\,32$ and $\mathsf{eq}\,33$ in $q_0 \to q_2$).

Relation to Register Automata. There is a natural connection between TOPL automata and *Register Automata* [21,24]. In particular, register automata are TOPL automata with $n = 1$ and labels from $\Lambda_{\mathrm{R}} \subseteq \Lambda$, where

$$\Lambda_{\mathrm{R}} = \{\, (\mathsf{fresh}, \mathsf{set}\,i) \mid i \in \{1, ..., m\} \,\} \cup \{\, (\mathsf{eq}\,i, \mathsf{nop}) \mid i \in \{1, ..., m\} \,\}$$

and $\mathsf{fresh} \equiv (\mathsf{neq}\,1 \text{ and } \mathsf{neq}\,2 \text{ and } \cdots \text{ and } \mathsf{neq}\,m)$.[3] In fact, we can show that the restrictions above are not substantial, in the sense that TOPL automata are reducible to register automata, and therefore equally expressive. In the following statement we use the standard injection $f : (V^n)^* \to V^*$ such that $f(\mathcal{L}(\mathcal{A})) = \{\, v_1^1 \ldots v_1^n \cdots v_k^1 \ldots v_k^n \mid (v_1^1, \ldots, v_1^n) \cdots (v_k^1, \ldots, v_k^n) \in \mathcal{L}(\mathcal{A}) \,\}$.

Proposition 5 (TOPL to RA). *There exists an algorithm that, given a TOPL automaton $\mathcal{A}$, builds a register automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = f(\mathcal{L}(\mathcal{A}))$. If $\mathcal{A}$ has m registers, $|\delta|$ transitions, $|Q|$ states, and works over n-tuples, then $\mathcal{A}'$ has $2m + 1$ registers, $|\delta'| = O(n(2m)^{2m}|\delta|)$ transitions and $O((2m)^m|Q| + |\delta'|)$ states.*

[3] Here a register automaton corresponds directly to what in [24] is called a 1N-RA.

High-level Automata TOPL automata seem to be lacking the convenience one would desire for verifying actual programs. In particular, when writing down a monitor for a specific violation, one may naturally not want to specify all other possible behaviours of the program (which may, though, be of relevance to other monitors). In fact, program behaviours not relevant to the violation under consideration can be *skipped*, ignored altogether. A possible solution for the latter could be to introduce loops with empty guards and actions, with the hope to consume the non-relevant part of the program behaviour. However, such a solution would not have the desired effect: the empty loops could also consume input relevant to the monitored violation.

The above considerations lead us to introduce a new kind of automaton where inputs can be skipped. That is, at each configuration x of a such an automaton, if an input does not match any of the guards of the available transitions from x the automaton will skip that input and examine the next one. In order to accommodate cases where we want specific transitions to happen consecutively, without skipping in between, we allow our automata to operate on sequences of letters, rather than single ones.

Definition 6. *A* ***high-level TOPL automaton (hl-TOPL)*** *is a tuple* $\mathcal{A} = \langle Q, q_0, s_0, \delta, F\rangle$ *where:*
- *Q is a finite set Q of states, with initial one $q_0 \in Q$ and final ones $F \subseteq Q$;*
- *$s_0 \in S$ is an initial store;*
- *$\delta \subseteq Q \times \Lambda^* \times Q$ is a finite transition relation.*

Although the definition of the syntax of high-level automata is very similar to that of ordinary TOPL automata, their semantics is quite different. A ***high-level configuration (hl-configuration)*** is a pair (x, w) of a configuration x and a word w; we denote the set of hl-configurations by $Y = X \times \Sigma^*$. We think of w as *yet to be processed.* A hl-configuration is *initial* when its configuration is the initial configuration; that is, it has the shape $((q_0, s_0), w)$. A hl-configuration is *final* when its state is final and its word is the empty word; that is, it has the shape $((q, s), \epsilon)$, where $q \in F$ and ϵ is the empty word. The hl-configuration graph is a subset of $Y \times \Sigma^* \times Y$. We write $y_1 \overset{w}{\hookrightarrow}_{\mathcal{A}} y_2$ to mean that (y_1, w, y_2) is in the hl-configuration graph of $\mathcal{A}$.

The following concept simplifies the definition of the hl-configuration graph. For each store s and sequence of pairs (g_i, a_i) of guards and actions ($i = 1, \ldots, d$), we construct a TOPL automaton

$$\mathcal{T}\big(s, (g_1, a_1), \ldots, (g_d, a_d)\big)$$

with set of states $\{0, \ldots, d\}$, out of which 0 is initial and d is final, initial store s, and transitions $(i{-}1, (g_i, a_i), i)$ for each $i = 1, \ldots, d$. Recall that, in this case, $\ell_1 \ldots \ell_d$ is accepted by the automaton when there exist configurations $x_0, x_1, \ldots, x_d$ such that $x_0 = (0, s)$ and $x_{i-1} \xrightarrow{\ell_i} x_i$, for each $i = 1, \ldots, d$. If the store of x_d is s' we say that the automaton can accept $\ell_1 \ldots \ell_d$ with store s'.

Definition 7. *The* ***configuration graph*** *of a hl-TOPL automaton $\mathcal{A}$ consists of two types of transitions:*
- Standard transitions, *of the form* $((q_1, s_1), ww') \overset{w}{\hookrightarrow} ((q_2, s_2), w')$, *when there exists $(q_1, \bar{\lambda}, q_2) \in \delta$ such that $\mathcal{T}(s_1; \bar{\lambda})$ can accept w with store s_2.*

– Skip transitions*, of the form* $(x, \ell w) \overset{\ell}{\hookrightarrow} (x, w)$*,*
when no standard transition starts from $(x, \ell w)$*.*

The ***language*** $\mathcal{L}(\mathcal{A})$ *of* $\mathcal{A}$ *is the set of words that label paths from an initial hl-configuration to a final one:* $\mathcal{L}(\mathcal{A}) = \{\, w_1 \dots w_k \mid y_0$ *initial,* y_k *final,* $\forall i \leq k.\, y_{i-1} \overset{w_i}{\hookrightarrow} y_i \}$.

Remark 8. Note that a TOPL automaton $\mathcal{A}$ can be technically seen as a high-level one with singleton transition labels. However, its language is in general different from the one we would get if we interpreted $\mathcal{A}$ as a high-level machine. For example, let $\mathcal{A}$ be the TOPL automaton consisting of one transition labelled with the guard $\mathsf{eq}\, 1$, from the initial state to the final state. The alphabet is $\Sigma = V$ and the initial store has one register containing value v. The language of $\mathcal{A}$ consists of one word made of one letter, namely v. On the other hand, because of skip transitions, the language of $\mathcal{A}$ seen as a hl-TOPL automaton consists of all words that contain the letter v.

Example 9. Consider the following hl-TOPL automaton with 2 registers over the alphabet $\Sigma = V = \{A, B\}$.
– $Q = \{1, 2, 3\}$, $q_0 = 1$ and $s_0 = (A, B)$ and $F = \{3\}$,
– δ consists of $\big(1, \big[(\mathsf{eq}\, 2, \mathsf{nop}), (\mathsf{eq}\, 1, \mathsf{nop}), (\mathsf{eq}\, 2, \mathsf{nop})\big], 2\big)$ and $\big(1, \big[(\mathsf{eq}\, 1, \mathsf{nop})\big], 3\big)$.
The language of this is automaton consists of those words in which the first A is not surrounded by two Bs.

We next present transformations between the two different classes of automata we introduced. First, we can transform TOPL automata to high-level ones by practically disallowing skip transitions: we obfuscate the original automaton $\mathcal{A}$ with extra transitions to a non-accepting sink state, in such a way that no room for skip transitions is left.

Proposition 10 (TOPL to hl-TOPL). *There exists an algorithm that, given a TOPL automaton* $\mathcal{A}$ *with* $|Q|$ *states, at most* d *outgoing transitions from each state, and guards with at most* k *conjuncts, it builds a hl-TOPL automaton* $\mathcal{A}'$ *with* $|Q| + 1$ *states and at most* $(d + k^d)|Q|$ *transitions such that* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$*.*

The converse is more difficult. A TOPL automaton simulates a given hl-TOPL one by delaying decisions. Roughly, there are two modes of operation: (1) store the current letter in registers for later use, and (2) simulate the configuration transitions of the original automaton. The key insight is that Step 2 is entirely a static computation. To see why, a few details about Step 1 help.

The TOPL automaton has registers to store the last few letters. The states encode how many letters are saved in registers. The states also encode a repartition function that records which TOPL register simulates a particular hl-TOPL register or a particular component of a past letter. The repartition function ensures that distinct TOP registers hold distinct values. Thus, it is possible to perform equality checks between hl-TOPL registers and components of the saved letters using only the repartition function. Similarly, it is possible to simulate copying a component of a saved letter into one of the hl-TOPL registers by updating the repartition function. Because it is possible to evaluate guards and simulate actions statically, the run of the hl-TOPL automaton can be completely simulated statically for the letters that are saved in registers.

Proposition 11 (hl-TOPL to TOPL). *There exists an algorithm that, given a hl-TOPL automaton $\mathcal{A}$, builds a TOPL automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. If $\mathcal{A}$ is over the alphabet V^n with m registers, $|Q|$ states, and $|\delta|$ transitions of length $\leq d$, then $\mathcal{A}'$ is over the alphabet V with $m' = m + (d-1)n$ registers, $O(d^2(m+1)^m|Q|)$ states, and $O(d^2(m+1)^{(m+n)}|\delta|)$ transitions.*

Remark 12. Although Propositions 10 and 11 imply that hl-TOPL and TOPL automata are equally expressive, the transformations between them are non-trivial and substantially increase the size of the machines (especially in the hl-TOPL-to-TOPL direction). This discrepancy is explained by the different goals of the two models: TOPL automata are meant to be easy to analyse, while high-level automata are meant to be convenient for specifying properties of object-oriented programs. The runtime monitors implement the high-level semantics directly.

Since both TOPL and hl-TOPL automata can be reduced to register automata, using known results for the latter [24] we obtain the following.

Theorem 13. *TOPL and hl-TOPL automata share the following properties.*
1. *The emptiness and the membership problems are decidable.*
2. *The language inclusion, the language equivalence and the universality problems are undecidable in general.*
3. *The languages of these automata are closed under union, intersection, concatenation and Kleene star.*
4. *The languages of these automata are not closed under complementation.*

The first point of Theorem 13 guarantees that monitoring with TOPL automata is decidable. On the other hand, by the second point, it is not possible to automatically validate refactorings of TOPL automata. Closure under regular operations, apart from negation, allows us to write specifications as negation-free regular expressions. The final point accentuates the difference between property violation and validation.

4 TOPL Properties

In this section we describe the user-level *Temporal Object Property Language (TOPL)*, which provides a programmer-friendly way to write down hl-TOPL automata relevant to runtime verification. The full syntax of the language was presented in [18]. Below we give the main ingredients and define the translation from the language to our automata.

A TOPL property comprises a sequence of ***transition statements***, of the form

```
source -> target: label
```

where source and `target` are identifiers representing the states of the described automaton. The sequence of statements thus represents the transition relation of the automaton. Each property must include distinguished vertices `start` and `error`, which correspond to the initial and (unique) final states respectively.

The set of labels has been crafted in such a way that it captures the observable events of program executions. Observable events for TOPL properties are method calls and returns, called ***event ids***, along with their parameter values. The set of event ids is given by the grammar:

$$E \ ::= \ \mathit{call}\ m \mid \mathit{ret}\ m$$

where m belongs to an appropriate set of ***method names***. Each method name has an *arity*, which we shall in general leave implicit. The set V_L of possible parameter values is a set of values specified by the programming language (e.g. Java) plus a dummy value $\bot$. The set of all values is $V = V_L \cup E$.

Labels of TOPL properties refer to registers via ***patterns***. A register v is called a ***property variable*** and has three associated patterns:

- the uppercase pattern V matches any value and writes it in the property variable v;
- the lowercase pattern v reads the value of the property variable v and only matches that value; and
- the negated lowercase pattern !v reads the value of the property variable v and only matches different values.

In addition, every element of V acts as a pattern that matches only the value it denotes, and a wildcard (*) pattern matches any value. The set of all patterns is denoted by Pat. A transition label can take one of the three forms:

$$l \ ::= \ \mathit{call}\ m(x_1, \ldots, x_k) \ \mid \ \mathit{ret}\ x := m \ \mid \ x := m(x_1, \ldots, x_k)$$

where $x, x_1, \ldots, x_k \in \mathit{Pat}$. Note that the latter two forms are distinct – the last one incorporates a call and a matching return. Finally, a TOPL property is *well-formed* when it satisfies the conditions:

(i) each label must contain an uppercase value pattern at most once;
(ii) any use of a lowercase pattern (i.e. a read) must be preceded by a use of the corresponding uppercase pattern (i.e., a write) on all paths from `start`.

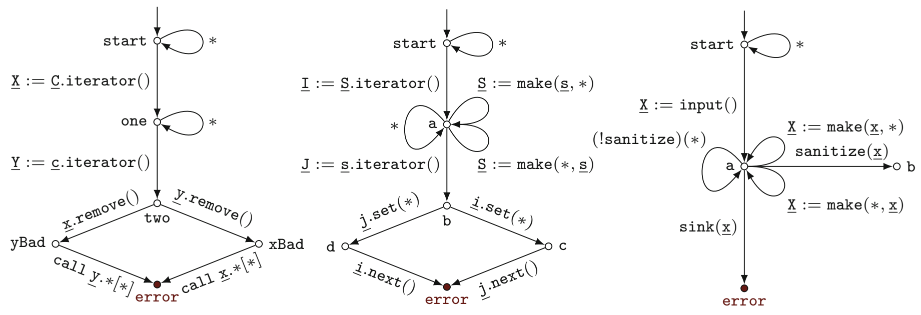
From now on we assume TOPL properties to be well-formed.

From TOPL to automata. We now describe how a TOPL property P yields a corresponding hl-TOPL automaton $\mathcal{A}_P$. First, if n is the maximum arity of all methods in P, the alphabet of $\mathcal{A}_P$ will be:

$$\Sigma_P = E \times V_L^{n+1}$$

where the extra register is used for storing return values. Note that Σ_P follows our previous convention of alphabets: it is a sub-alphabet of $\Sigma = V^{n+2}$. For example, if P has a maximal arity 5, the event $\mathit{call}\ m(a, b, c)$ would be understood as $(\mathit{call}\ m, \bot, a, b, c, \bot, \bot)$ by $\mathcal{A}_P$. Here the first component is the event id, the second is a filler for the return value, the next three are the parameter values and the rest are paddings which are used in order for all tuples to have the same length. The event $\mathit{ret}\ r = m$ would be understood as $(\mathit{ret}\ m, r, \bot, \bot, \bot, \bot, \bot)$ by $\mathcal{A}_P$.

We include in $\mathcal{A}_P$ one register for each property variable in P and, in order to match elements from V, we include an extra register for each element mentioned by P (this includes all the method names of P). Each extra register contains a specified value in the initial state of $\mathcal{A}_P$ and is never overwritten. The rest of the registers in the initial state are empty. We write Pat_P for the set of patterns of property variables appearing in P.



(a) Property 2 (b) Property 3 (c) Property 4

Fig. 3. TOPL formalisations of the example properties from Section 2

We next consider how labels are translated. The first two forms of label ($call$ and ret) describe observable events and are translated into one-letter transitions in $\mathcal{A}_P$, while the latter form is translated into two-letter transitions. Let $\{1, \ldots, N\}$ be the set of registers of $\mathcal{A}_P$. We define three functions: $reg : Pat_P \to (N \cup \{\bot\})$ associates a register to each pattern (with $reg(*) = \bot$), while $grd : Pat_P \times N \to G$ and $act : Pat_P \times N \to A$ give respectively the guard and action correspoding to each pattern and register. We set:

$$grd(x,j) = \begin{cases} \mathsf{true} & \text{if } x = \mathtt{V} \\ \mathsf{eq}\, reg(\mathtt{v})\, j & \text{if } x = \mathtt{v} \\ \mathsf{neq}\, reg(\mathtt{v})\, j & \text{if } x = !\mathtt{v} \\ \mathsf{eq}\, reg(x)\, j & \text{if } x \in V \\ \mathsf{true} & \text{if } x = * \end{cases} \qquad act(x,j) = \begin{cases} \mathsf{set}\, reg(\mathtt{v}) := j & \text{if } x = \mathtt{V} \\ \mathsf{nop} & \text{if } x = \mathtt{v} \\ \mathsf{nop} & \text{if } x = !\mathtt{v} \\ \mathsf{nop} & \text{if } x \in V \\ \mathsf{nop} & \text{if } x = * \end{cases}$$

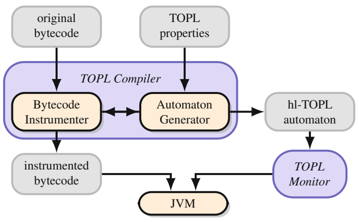
We can now interpret labels of P into labels of $\mathcal{A}_P$. For each a label l of P, we define its translation $[\![l]\!] = [([\![l]\!]_G, [\![l]\!]_A)]$, where $[\![-]\!]_G$ and $[\![-]\!]_A$ are given as follows.

$$[\![l]\!]_G = \begin{cases} grd(m,1)\ \mathsf{and}\ grd(x_1,3)\ \mathsf{and} \ldots \mathsf{and}\ grd(x_k,k{+}2) & \text{if } l = call\ m(x_1,\ldots,x_k) \\ pred(m,1)\ \mathsf{and}\ \mathsf{eq}\, reg(x)\, 2 & \text{if } l = ret\ x := m \end{cases}$$

$$[\![l]\!]_A = \begin{cases} act(x_1,3)\ \mathsf{and} \ldots \mathsf{and}\ act(x_k,k{+}2) & \text{if } l = call\ m(x_1,\ldots,x_k) \\ act(x,2) & \text{if } l = ret\ x := m \end{cases}$$

Finally, for the label $x := m(x_1, \ldots, x_k)$, observe its right-hand-side refers to a call, while its left-hand-side refers to a return. We take this label to mean that m is called with parameters matching $x_1, \ldots, x_k$ and returns a value matching x, *and no event is observed in the meantime*. This is because an intermediate call, for instance a recursive call, could disconnect the method call and the return value. Thus, this label translates into a transition of length two:

$$[\![x := m(x_1,\ldots,x_k)]\!] = [\![call\ m(x_1,\ldots,x_k)]\!]\, [\![ret\ x := m]\!]$$



	Number of tracked active configurations			
	reference	≤ 0	$\leq 10^1$	$\leq 10^2$
tomcat	5.3±0.1	5.4±0.1	5.6±0.2	9.0±0.3
pmd	5.2±0.4	5.4±0.2	12.2±0.3	47.7±10.7
h2	6.6±0.2	9.5±0.2	130.1±12.2	timeout

Fig. 4. Left: Architecture of the TOPL tool. **Right:** Experimental Results. Times are in seconds, averaged over 10 runs (not in convergence mode).

Examples. Figure 3 displays the formal versions of the first three properties that are discussed in Section 2.

(a) This example illustrates how multiple related objects are tracked. In state `two`, the property tracks all pairs of two iterators x and y for the same collection c. If x.`remove`() is called, then state `yBad` becomes active, which precludes further use of y's methods. State `xBad` is symmetric.

(b) This example illustrates how chaining of values is tracked, while at the same time tracking multiple related objects. Recall that Property (3) refers to the code in Figure 2 (on page 263). In state `a`, the iterator i refers to the string s or some substring of s. In state `b`, the itrerator j refers to s. As opposed to the previous property, the two iterators i and j are not necessarily for the same collection, but rather for a collection and one of its sub-collections. This property does not refer to the Java standard library, which does not implement ropes. There exist, however, several independent libraries that follow the pattern in Figure 2 (e.g. `http://ahmadsoft.org/ropes/`).

(c) This example illustrates sanitization of values, in addition to chaining. In state `a`, the property keeps track of the tainted object x. An object is *tainted* if it comes from a specific input method or was made from tainted objects, and was not sanitized. A tainted object must not be sent to a `sink`.

Of course, the input of the TOPL compiler is not in graphical form. Below we include the actual representation for a property of type (c) without the sanitization option. It specifies actual methods that provide tainted inputs, make tainted objects out of tainted objects, and constitute sinks.[4] We refer the reader to [18] for more example properties.

```
property Taint
  prefix <javax.servlet.http.HttpServletRequest>
  prefix <java.lang.String>
  prefix <java.sql.Statement>
  start -> start:       *
  start -> tracking:    X := *.getParameter[*]
  tracking -> tracking: *
  tracking -> tracking: X := x.concat(*)
  tracking -> tracking: X := *.concat(x)
  tracking -> error:    *.executeQuery(x)
```

[4] Note that, to ease the task of writing TOPL properties, we have included a `prefix` directive: `prefix` p produces from every method name m, an extra name pm; it further produces, from any transition involving m, a similar transition involving pm.

5 Implementation and Experiments

The TOPL tool[5] checks at runtime whether Java programs violate TOPL properties. It consists of a compiler and a monitor (see Figure 4, left).

Given the bytecode of a Java project and several TOPL properties, the compiler produces instrumented bytecode and a hl-TOPL automaton. An instrumented method emits a call event, runs the original bytecode, and then emits a return event. Emitting an event is encoded by a call to the method `check(Event)` of the monitor. The `Event` structure contains an integer identifier and an array of `Objects`. The identifier is unique for each site from which the method `check` is called. The compiler achieves two tasks that are interdependent: instrumenting the bytecode, and translating properties into an automaton. The instrumentation could be done on all methods of the Java project, but this would lead to high runtime overhead. Instead, the compiler instruments only the methods that are mentioned by the TOPL properties to be checked. Conversely, the translation of properties into automata depends on the Java project's code. To see why, consider a transition guarded in a property by the method name pattern m. The compiler instruments all the methods whose (fully qualified) names match the pattern m, and all the methods that override methods whose names match the pattern m, thus taking into account inheritance. All these instrumented methods emit events with identifiers from a certain set of integers, which depends on the inheritance structure of the Java project. The method name pattern m is essentially compiled into a set of integer event identifiers.

The monitor is an interpreter for the hl-TOPL automaton that the compiler produces. Its implementation closely follows the semantics from Section 3. For example, the monitor maintains a set of active configurations, which are those reachable by a path labeled by the events seen so far. There are, however, several differences. First, the number of active configurations is not bounded in theory, but a bound may be enforced in practice. Monitoring becomes slower as the number of active configurations increases. As a pragmatic compromise, the user may impose an upper bound, thus trading soundness for efficiency. That is, if the user imposes a bound then monitoring is faster, but property violations may be missed (on the other hand, a reported violation of a property is always a real violation). Second, the implementation includes several optimizations. For example, the guards produced by method name patterns, which require the current event id to be from a certain set of integers, is implemented as a hashtable lookup rather than as a linear search, as the formal semantics would suggest. Third, the implementation saves extra information in order to provide friendlier error messages. For instance, the user may ask the TOPL monitor to save and report the path taken in the configuration graph, or full call stack traces for each event.

Experimental Results. We measured the overhead on the test suite DaCapo [13], version 9.12. DaCapo is a collection of automated tests that exercise large portions of code from open-source projects and the Java standard libraries. DaCapo itself has been used for many experiments by the research community. Hence, we did not expect to find any bugs, but aimed instead at measuring the overhead. We checked two types of properties with TOPL. First, properties that express correct usage of the standard Java libraries.

[5] `http://rgrig.github.com/topl`

Table 1. Experiment on small properties (taken from [25]) run on the DaCapo benchmarks (in convergence mode). HasNext checks that no iterator is advanced without first enquiring hasNext. UnsafeIterator checks that no iterator is advanced after the iterated collection has been modified. UnsafeMapIterator checks that no iterator on keys/values of a map is advanced after the map has been updated. UnsafeFileWriter checks that no file is written to after it was closed. Column original gives the running times (in seconds) for projects without instrumentation of Java standard libraries. The other columns report instrumented runs, with a maximum of 3 and 10 active configurations.

	original	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeFileWriter	
		st=3	st=10	st=3	st=10	st=3	st=10	st=3	st=10
avrora	8.1	27.8	60.5	163.3	323.1	194.5	179.9	8.3	5.9
batik	1.2	18.1	3.0	3.8	3.8	3.1	3.3	1.3	1.2
eclipse	17.4	24.2	24.0	30.9	41.7	27.2	28.0	22.9	22.8
fop	0.3	0.9	1.9	3.5	3.6	2.7	2.7	0.3	0.3
h2	6.2	5.9	6.8	8.3	20.0	13.5	11.2	6.4	6.0
jython	1.9	19.8	46.1	81.5	83.0	62.8	62.7	1.9	1.8
luindex	0.8	0.8	0.8	0.8	0.9	1.0	0.9	0.8	0.9
lusearch	1.5	1.5	1.5	15.0	16.0	13.8	12.8	1.5	1.7
pmd	3.1	19.9	42.6	93.5	240.3	102.6	105.6	3.2	3.3
sunflow	3.9	3.8	3.9	4.0	3.8	3.9	3.9	3.9	4.3
tomcat	2.5	4.2	8.3	22.9	50.9	30.0	31.0	2.6	2.7
xalan	1.5	14.5	7.1	425.0	360.9	272.0	276.5	1.5	1.2

Second, properties that express temporal constraints which we extracted from the code comments of three open-source projects (H2, PMD, Tomcat) included in DaCapo. H2 is a database server for which we checked properties on the calling order of some interface methods. For example, a client should not attempt to ask for a row from a cursor unless the latter has been previously advanced. PMD looks for bugs, dead code and other problems in Java code. One of the five properties we checked is "Only if a scope replies that it knows a name, it can be asked for that name's definition". Tomcat is a highly concurrent servlet server. Servlets are Java programs running in a webserver, extracting data from `ServletRequests` and sending data to `ServletResponses`. A response has two associated incoming channels: a stream and a writer. They should not be both used concurrently. But the servlet, before forwarding the response, must call `flush` on the stream, on the writer, or on the response itself. This is one of the properties we checked. Interestingly, while experimenting with Tomcat, TOPL discovered a concurrency bug (a data race) in the DaCapo's infrastructure which would manifest sometimes as `null` dereference.

Although our tool is not currently optimised, we measured both time and space overhead. It turns out that space overhead is negligible, below the variance caused by the randomness of garbage collection. Thus, we only report on time overhead, in Figure 4 and Table 1. The relative overhead is meaningful only if the reference runtime is not close to 0, and this is most distinctively the case for test eclipse whose runtime is over 10 s. The (geometric) average overhead in that case is $\times 1.5$ with $\leq$ 3 active configurations, and $\times 1.6$ with $\leq$ 10 active configurations. Figure 4 shows the effect of tuning the active configurations in terms of overhead. All experiments were performed on an Intel i5 with 4 cores at 3.33 GHz with 4 GiB of memory, running Linux 2.6.32 and Java VM 1.6.0_20.

6 Related Work

JavaMOP [23] and *Tracematches* [2] are based on slicing: A slice is a projection of a word over a finite alphabet; different slices are fed, independently, to machines that handle finite alphabets. Tracematches use regular expressions to specify recognisers over finite alphabets. JavaMOP supports several other logics, via a plugin mechanism, and slices are assigned categories, which can be match/fail or taken from some other set. Because slices are analyzed independently, it not possible to express examples such as (1) and (4), which use an unbounded number of register assignments.

Quantified Event Automata (QEA) [6] extend the slicing mechanism of JavaMOP with the goal of improving expressivity. Similarly to TOPL automata, QEAs have guards and assignments, which can be arbitrary predicates and transformations of the memory content respectively. In contrast, our automata impose specific restrictions, which follow the expressive power of RAs. In addition, QEAs introduce quantifiers, which can be seen as a way to impose a hierarchy on slices. Systems based on machines with parametric transition rules, such as *RuleR* [9], *LogScope* [7] and *TraceContract* [8], are related to QEAs and are also very similar in spirit to the TOPL approach. RuleR is tuned towards high expressivity and in particular can handle context-free grammars with parameters, which go beyond the reach of TOPL. By comparison, TOPL automata seem a simpler formalism, and this paper demonstrates how they are closely related to standard automata-theoretical models.

QVM [3] is a runtime monitoring approach tailored to deployed systems. It achieves high efficiency by being carefully implemented inside a Java virtual machine, checking properties involving a single object, and being able to tune its overhead on-the-fly. On the other hand, TOPL is designed for aiding the programmer during development and testing, and therefore focusses instead on providing a precise and expressive language for specifying temporal properties. For instance, TOPL can express properties involving many objects. Both QVM and TOPL let the programmer tune the overhead/coverage balance. *ConSpec* [1] is a language used to describe security policies. Although ConSpec automata have a countable number of states, they are deterministic and therefore cannot express the full range of TOPL properties.

From the techniques used mostly for static verification of object-oriented programs, *typestates* [29] are probably the most similar to TOPL. A modular static verification method for typestate protocols is introduced in [11]. The specification method is based on linear logic, and relations among objects in the protocol are checked by a tailored system of permissions. Similarly, [15,10] provide a means to specify typestate properties that belong to a single object. The specified properties are reminiscent of contracts or method pre/post-conditions and can deal with inheritance. In [17] the authors present sound verification techniques for typestate properties of Java programs, which we envisage that can be fruitfully combined with the TOPL paradigm. Their approach is divided into several stages each employing its own verifier, with progressively higher costs and precisions. Every stage focuses on verifying only the parts of the code that previous stages failed to verify.

A specification language for interface checking aimed at C programs, called *SLIC*, is introduced in [5]. SLIC uses non-determinism to encode universal quantification of dynamically allocated data and allows for complex code in the automaton transitions; while TOPL specifications naturally express universally quantified properties over data structures and, for effectiveness reasons, there is a limit on the actions performed during automaton transitions. Simple SLIC specifications are verified by the SLAM verifier [4].

Similar investigations have been pursued by the functional programming community. In [16] contracts are used for expressing legal traces of programs in a functional language with references. The contracts specify traces as regular expressions over calls and returns, thus resembling our automata, albeit in quite a different setting. The specifications are function-centered and, again, capturing inter-object relations seems somewhat tricky.

Finally, as demonstrated in previous sections, TOPL automata are a variant of register automata [21,24], themselves a thread in an extensive body of work on automata over infinite alphabets (see e.g. [28]). RAs form one of the most well-studied paradigms in the field, with numerous extensions and variations (e.g. [14,12,28]).

Acknowledgements. We acknowledge funding from EPSRC H011749 (Distefano and Grigore) and RAEng research fellowships (Distefano and Tzevelekos). We thank the anonymous reviewers, for their suggestions significantly improved the paper.

References

1. Aktug, I., Naliuka, K.: ConSpec – a formal language for policy specification. ENTCS 197(1), 45–58 (2008)
2. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA, pp. 345–364 (2005)
3. Arnold, M., Vechev, M., Yahav, E.: QVM: an efficient runtime for detecting defects in deployed systems. In: OOPSLA, pp. 143–162 (2008)
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. Ball, T., Rajamani, S.K.: SLIC: a specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research (2002)
6. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012)
7. Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal Analysis of Log Files. Journal of Aerospace Computing, Information, and Communication 7(11) (2010)
8. Barringer, H., Havelund, K.: TRACECONTRACT: A Scala DSL for Trace Analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
9. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule Systems for Run-time Monitoring: from Eagle to RuleR. J. Log. Comput. 20(3), 675–706 (2010)
10. Bierhoff, K., Aldrich, J.: Lightweight object specification with typestates. In: ESEC/ SIGSOFT FSE, pp. 217–226 (2005)

11. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA, pp. 301–320 (2007)
12. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theor. Comput. Sci. 411(4-5), 702–715 (2010)
13. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA, pp. 169–190 (2006)
14. Bojanczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-Variable Logic on Words with Data. In: LICS, pp. 7–16 (2006)
15. DeLine, R., Fähndrich, M.: Typestates for Objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)
16. Disney, T., Flanagan, C., McCarthy, J.: Temporal higher-order contracts. In: ICFP, pp. 176–188 (2011)
17. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: ISSTA, pp. 133–144 (2006)
18. Grigore, R., Petersen, R.L., Distefano, D.: TOPL: A language for specifying safety temporal properties of object-oriented programs. In: FOOL (2011)
19. Havelund, K., Rosu, G.: Monitoring Programs Using Rewriting. In: ASE, pp. 135–143 (2001)
20. Jin, D., Meredith, P., Lee, C., Rosu, G.: JavaMOP: Efficient parametric runtime monitoring framework. In: ICSE, pp. 1427–1430 (2012)
21. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. 134(2) (1994)
22. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
23. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT 14(3), 249–289 (2012)
24. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Logic 5(3), 403–435 (2004)
25. Newsome, J., Song, D.X.: Dynamic Taint Analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: NDSS (2005)
26. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. LMCS 8(1) (2012)
27. Sakamoto, H., Ikeda, D.: Intractability of decision problems for finite-memory automata. Theor. Comput. Sci. 231(2), 297–308 (2000)
28. Segoufin, L.: Automata and Logics for Words and Trees over an Infinite Alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)
29. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. 12(1), 157–171 (1986)

Unbounded Model-Checking with Interpolation for Regular Language Constraints

Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte

The University of Melbourne
{ggange,jnavas,pjs,harald,schachte}@csse.unimelb.edu.au

Abstract. We present a decision procedure for the problem of, given a set of regular expressions $R_1, \ldots, R_n$, determining whether $R = R_1 \cap \cdots \cap R_n$ is empty. Our solver, REVENANT, finitely unrolls automata for $R_1, \ldots, R_n$, encoding each as a set of propositional constraints. If a SAT solver determines satisfiability then R is non-empty. Otherwise our solver uses unbounded model checking techniques to extract an interpolant from the bounded proof. This interpolant serves as an overapproximation of R. If the solver reaches a fixed-point with the constraints remaining unsatisfiable, it has proven R to be empty. Otherwise, it increases the unrolling depth and repeats. We compare REVENANT with other state-of-the-art string solvers. Evaluation suggests that it behaves better for constraints that express the intersection of sets of regular languages, a case of interest in the context of verification.

1 Introduction

Strings are ubiquitous in software. Many web applications, for example, construct database queries from user-provided strings. The rapid rise in the popularity of these applications and the proliferation of vulnerabilities to attacks such as SQL injection and cross-site scripting can explain a renewed interest in developing practical, efficient verification techniques for reasoning about strings.

Regular expressions are commonly used to define sanitization checks over strings. For example, a regular expression can be used as a filter to exclude strings that exhibit a particular attack pattern. Given a set of sanitization filters $F_1, \ldots, F_n$ and an attack pattern P, we wish to determine if $F_1 \cap \cdots \cap F_n \cap P$ is empty. Although this problem is decidable, the implementation of practical algorithms is still an open issue. Most state-of-the-art solutions (e.g., [22,9,11]) rely on the classical product algorithm for intersection of DFAs, but they differ in how they tackle the two main performance bottlenecks: exponential blowup while converting regular expressions to DFAs, and the large state space of the product automaton. These solvers, particularly lazy solvers [11], are very efficient when the query is underconstrained because they can avoid building the full product automaton. To prove unsatisfiability, however, they must enumerate the complete set of reachable product states. This is not desirable in the context of verification, where we may be testing the intersection of many languages (with a potentially exponential product automaton), and unsatisfiable queries are common.

In this paper, we develop an alternative approach for checking intersection of a set of regular expressions using established SAT-based unbounded model-checking techniques. We first translate the regular expressions $R_1, \ldots, R_n$ into a set of *SFAs (symbolic finite-state automata)* [21]. An SFA is a generalization of a finite-state automaton where transitions are labelled with a symbolic encoding of a set of values, rather than requiring a separate transition for each value. Although the use of SFAs is not new, it is worth mentioning that our method does not require any determinization of the SFAs. Next, we unroll each SFA up to a fixed depth k, encode each unrolled SFA as a set of propositional constraints, and use a SAT solver to determine satisfiability. This encoding consists mainly of the conjunction of the constraints originating from the initial states, transitions, and final states of each unrolled SFA. If the constraints are satisfiable then we return a string w that belongs to the intersection of the languages as a witness. Otherwise, we have proven that the intersection is empty for strings up to length k. However, this is not sufficient to prove that the intersection is empty in the unbounded case. To overcome this, we apply McMillan induction [16]. The idea is to use *interpolation* [5] to generalize a proof for the length-k case to one that proves the intersection empty for any length. In summary:

- We address the unbounded model checking problem as applied to string solving; unlike other “unbounded” methods, we combine SAT solving with the interpolation-based approach of McMillan [16], instantiating that framework to the case of SFA unrolling.
- We describe REVENANT, a publicly available solver designed to handle the intersection of *sets* (beyond *pairs*) of regular languages efficiently.
- We compare with the state-of-the-art solvers REX [22], DPRLE [9], and STRSOLVE [11], using a standard benchmark set of regular expressions extracted from real applications [21], together with intersection instances designed to stress test solvers. REVENANT performs very well on instances in its target domain, while remaining competitive across benchmarks.

2 Related Work

Methods for solving language constraints can loosely be divided into bounded and unbounded methods.

Bounded methods (e.g., HAMPI [14], KUDZU [20], and CFGANALYZER [1]) unroll the constraints to a given length bound, encode the unrolled problem as a set of propositional formulas, and use a SAT solver to determine satisfiability. These methods can be quite efficient finding a satisfying assignment and often can express a wider range of constraints than the unbounded methods. However, if unsatisfiability results then no useful conclusions can be derived. Thus, these tools are not suitable for verification, which is our main motivation.

Existing unbounded methods instead build the classical decision procedures. Wasserman *et al.* [23] build on ideas by Minamide [18] to overapproximate string variables with context-free grammars and model a potential SQL attack with a finite automaton. They build the product of a push-down automaton, constructed from the context-free grammar, with the finite automaton that captures a potential SQL attack, and check if the language of the resulting automaton is empty. Rex [22] improves upon the classical FSA algorithms by introducing *symbolic* finite-state automata (SFAs), where each edge is annotated with a *set* (in the form of a one-place predicate), rather than a single symbol. Rex then uses the SMT solver Z3 [6] to manipulate edge constraints during operations such as intersection and determinization. Efficiency is achieved by keeping SFAs "clean" (avoiding unsatisfiable formulas as edge labels on moves). Hooimeijer *et al.* [9] present dprle which also relies on the classical algorithms for regular languages involving concatenation and subset constraints. dprle utilises dependency analysis information to slice away product automaton states that are irrelevant for the query. The same authors have later developed a lazy solver called StrSolve [11] which outperforms previous approaches. StrSolve performs a lazy search space enumeration by considering only those states from the product automata needed for the query.

While our method falls in the "unbounded" class, we differ from previous approaches in our use of McMillan induction. As mentioned, our work can be seen as an application of McMillan's interpolation-based framework [16].

3 Unbounded Model Checking with Interpolation

Consider an unsatisfiable set F of Boolean formulas which has been partitioned into two sets A and B. An *interpolant* [5] of A and B is a formula P containing only variables that are common between A and B, and satisfying the properties

$$\begin{gathered} A \models P \\ P \wedge B \models \bot \end{gathered}$$

It is well known that, given an unsatisfiability proof for $A \wedge B$, an interpolant P can be generated in linear time [19,17].

The use of interpolants for SAT-based model checking was pioneered by McMillan [16]. SAT-based unbounded model-checking is formulated in terms of a transition system $T = (S, I, \delta, F)$, with a set of state variables S, initial conditions I, transition relation δ and final conditions F. A propositional encoding is constructed for the given transition system unrolled to depth k, and is tested for satisfiability. If the finite unrolling is satisfiable, we have produced a concrete error trace. Otherwise, we can generate an interpolant in accordance with the partitioning shown in Fig. 1. Note that $A = I \wedge \delta_0$ represents the set of T states reachable in one step from the initial conditions. Since the interpolant P is expressed in terms of state variables s_1 (the only variables shared by A and B), and satisfies the property $I \wedge \delta_0 \models P$, P is an overapproximation of states reachable in one step from the initial state. Now, by replacing each variable from s_1 in P by the corresponding variable from s_0, an over-approximation $P[s_0/s_1]$ of the reachable states is obtained, according to which F is still unreachable. If

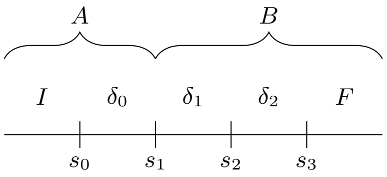


Fig. 1. The partitioning used for interpolant generation. Note that the only variables shared between A and B are s_1.

$P[s_0/s_1] \models I$, our initial conditions encompass all the reachable states; and since F is still unreachable, it must remain so after unrolling to any depth. If not, we can relax the initial condition to $I \vee P[s_0/s_1]$ and repeat the process from there. Eventually, either the relaxation will fail to weaken the initial condition, that is, the condition reaches a fixed-point, in which case we have proven unsatisfiability in the unbounded case, or the conjunction of constraints becomes satisfiable, in which case we must perform a longer unroll. This process is guaranteed to terminate [16].

4 Regular Language Representations

We now describe how regular languages are represented as symbolic finite-state automata, and how we manipulate these. We consider a simple constraint language given by the following grammar:

$$\begin{array}{ll} \mathit{Constraint} & \rightarrow \mathit{Var} \in \mathit{RegExp} \\ \mathit{RegExp} & \rightarrow \mathit{Lit} \mid \mathit{RegExp} + \mathit{RegExp} \mid \mathit{RegExp}\ \mathit{RegExp} \mid \mathit{RegExp}^* \end{array}$$

The only possible constraints are membership queries. *Lit* is the set of string literals. Intersection between regular expressions $R_1, \ldots, R_n$ can be expressed via the constraints $x \in R_1$, ..., $x \in R_n$. For convenience, our implementation supports other standard constructions such as ranges, bounded repetitions, special characters (\d, \w, and so on) which are made to conform with the grammar in a preprocessing step.

4.1 Symbolic Finite State Automata

Formally, a finite-state automaton is defined by a tuple $(Q, \Sigma, \delta, q_0, F)$. The automaton begins in state $q_0 \in Q$; at each step, the state is updated according to the transition relation δ. The automaton is said to *accept* if, at the end of input, it is in a state $q_i \in F$.

In a typical finite-state automaton, each edge is expressed as a triple (q_s, α, q_e), with $q_s, q_e \in Q$ and $\alpha \in \Sigma$. A *symbolic* finite-state automaton [22] extends this by encoding the edge as (q_s, ψ, q_e), where $\psi \subseteq \Sigma$ encodes the set of input values permitted by the transition. A number of encodings have been proposed for these sets of values, including hash-sets, range predicates and bit-vector constraints; these are discussed in [8].

Given that we wish to construct a propositional encoding of the automaton, we also require an encoding that can be conveniently transformed into a propositional formula, in addition to providing efficient construction and a concise encoding of value sets. Accordingly, we construct binary decision diagrams over the bit-vector encoding of the characters.

4.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are often used to represent Boolean functions. *BDD expressions* are defined inductively:

– $\mathcal{F}$ and $\mathcal{T}$ are BDD expressions.
– If x is a variable and e_1 and e_2 are BDD expressions then $\mathsf{ite}(x, e_1, e_2)$ is a BDD expression.

The *meaning* of a BDD expression is defined:

$$\begin{aligned}
[\![\mathcal{F}]\!] &= \mathit{false} \\
[\![\mathcal{T}]\!] &= \mathit{true} \\
[\![\mathsf{ite}(x, e_1, e_2)]\!] &= (x \wedge [\![e_1]\!]) \vee (\neg x \wedge [\![e_2]\!])
\end{aligned}$$

BDDs are the directed acyclic graphs that result when sub-expressions are allowed to be shared.

An *ordered* BDD assumes that variables are ordered by a linear order relation $\prec$. A BDD is an OBDD iff, whenever it is of form $\mathsf{ite}(x, e_1, e_2)$, e_1 and e_2 are OBDDs and each x' occurring in e_1 or e_2 satisfies $x \prec x'$. An OBDD e is *reduced* (and is called an *ROBDD*) iff $[\![\cdot]\!]$ is injective across e, that is, for all BDDs e_1 and e_2 appearing in e, $[\![e_1]\!] \equiv [\![e_2]\!] \Rightarrow e_1 = e_2$.

While the size of BDDs may be exponential in the number of variables, limited unions of character ranges can be concisely represented, as illustrated in Fig. 2. In these diagrams, $\mathsf{ite}(x, e_1, e_2)$ is captured by showing a solid arc from node x to the root of e_1 and a dashed arc from x to the root of e_2; except we omit the sink $\mathcal{F}$ and all arcs leading to it.

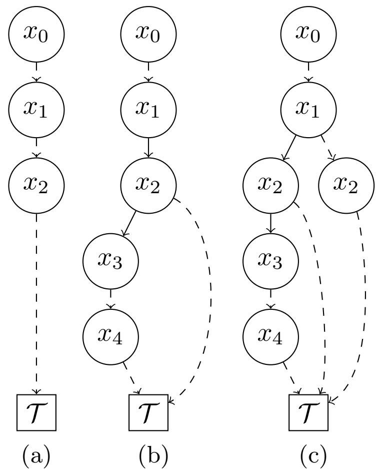


Fig. 2. BDDs that represent the 5-bit ranges (a) [0-3], (b) [8-12], and (c) the disjunction of the two

```
**sfa_reduce**$((Q, \Sigma, \delta, q_0, F))$
    $depend := (q \mapsto \{q' \mid (q', \psi, q) \in \delta, q' \neq q\})$
    **foreach** $q \in Q$ **do**
        $ufind$.make$(q)$
        $queue$.insert$(q)$
    $shash := \emptyset$
    **while**$(\neg queue$.empty$())$
        $q := queue$.pop$()$
        $q_m := ufind$.find$(q)$
        $dests := \{q \mapsto \bot \mid q \in Q\}$
        **for** $(q_s, \psi, q_d) \in \delta,$ such that $q_s = q_m$
            $dests(q_d) := \psi \vee dests(q_d)$
        $q_t := shash(\langle q_m \in F, dests\rangle)$
        **if**( $q_t \neq$ NotFound)
            **if**$(q_m \neq q_t)$
                $ufind$.merge$(q_m, q_t)$
                **foreach** $q' \in depend(q_m)$
                    $queue$.insert$(ufind$.find$(q'))$
                $depend(q_t) := depend(q_t) \cup depend(q_m)$
        **else**
            $shash(\langle q_m \in F, dests\rangle) := q_m$
    $Q' := \{q \in Q \mid ufind$.find$(q) = q\}$
    $q'_0 := ufind$.find$(q_0)$
    $\delta' := \emptyset$
    **foreach** $q'_i, q'_j \in Q'$
        $\alpha' := \bigvee \{\alpha \mid (q_i, \alpha, q_j) \in \delta, ufind$.find$(q_i) = q'_i, ufind$.find$(q_j) = q'_j\}$
        $\delta' := \delta' \cup \{(q'_i, \alpha', q'_j)\}$
    $F' := \{q \in F \mid ufind$.find$(q) = q\}$
    **return** $(Q', \Sigma, \delta', q'_0, F')$
```

Fig. 3. Pseudo-code for SFA reduction

4.3 SFA Reduction

The standard construction of an NFA from a regular expression often introduces a considerable number of redundant and equivalent states. The approach taken by Rex is to give symbolic equivalents to the classical ϵ-elimination, determinization and DFA minimization algorithms.

Given that the size of a deterministic automaton is potentially exponential relative to the corresponding NFA, we would prefer to reduce the size of the non-deterministic SFA directly. While finding the minimum number of states for an NFA is PSPACE-hard, approaches have been presented [13,12] for reducing the size of an NFA directly.

We first eliminate ϵ transitions, following the procedure used by Rex. We then use a simple structural hashing approach to eliminate redundant states introduced during automaton construction. All states are initially assumed to be distinct, and we progressively merge pairs of states which have identical transition relations.

The pseudo-code for this is given in Fig. 3. *ufind* maintains the renaming of equivalent states, and can be efficiently implemented using a union-find data-structure. $depend(q_j)$ is the set of states that must be checked if state q_j is renamed; *queue* maintains the set of states that still need to be checked, and *shash* is used to check state equivalences. This approach is strictly weaker than the partition refinement of Ilie and Yu [13]; however, in the presence of symbolic edges, it avoids the need to test the intersection of large numbers of transitions.

5 Model Checking Formulation

We consider the problem of, given a set of regular expressions $R_1, \ldots, R_n$, determining whether the intersection $R_1 \cap R_2 \cap \cdots \cap R_n$ is empty. By converting each regular expression R_i into a SFA A_i, this can be reduced to determining whether there is a sequence $x \in \Sigma^* = x_1, \ldots, x_k$ of inputs that will leave every automaton in an accept state.

We can reformulate this as a transition system with state space $Q' = Q_1 \times \ldots \times Q_n$, initial state $q'_0 = \langle q^0_1, \ldots, q^0_n \rangle$, accepting states $F' = F_1 \times \ldots \times F_n$, and transition relation

$$\delta(\langle s_1, \ldots, s_n \rangle, x) = \langle \delta_1(s_1, x), \ldots, \delta_n(s_n, x) \rangle$$

where δ_i is the transition relation for A_i. We wish to determine if there is any reachable state of the form

$$\langle q_1, \ldots, q_n \rangle \in F' \qquad \text{(i.e., } \forall_{i \in \{1, \ldots, n\}}\ q_i \in F_i)$$

We can then apply the unbounded model-checking procedure to this revised formulation. The procedure is described in Fig. 4 and resembles the one described by McMillan [16]. The main differences are in how we unroll the SFAs and define the interpolation groups A and B in order to approximate the bounded proofs generated by the SAT solver. Fig. 4 gives a high level description of the method. The procedure **Intersection** takes as inputs the set of transition systems that represent all the automata to be intersected and a value k that represents the unrolling depth. The algorithm makes use of I, F, and the procedure **unroll** which are explained in Section 5.1. For now, suffice it to say that I and F denote the Boolean encoding of the initial states q'_0 and accepting states F', respectively. The procedure **unroll** unwinds the transition system up to depth k. For convenience, **unroll** can be called to return the layers from 0 to 1 and 1 to k separately, so as to simplify the formation of interpolation groups A and B.

If the procedure **Intersection** returns INCONCLUSIVE then we need to increase the value of k. Although the process will eventually terminate, judicious choice of the next k can speed up the convergence of the fixed-point significantly. Experimentally we have observed that a good choice the first time we get inconclusive results is to increase k to the maximum of the shortest accepting run from any state in a single automaton. After that, we increase k by doubling its value.

```
Intersection($\{T_1, \ldots, T_n\}, k$)
    // $T_1 \equiv \langle Q_1, \Sigma, \delta_1, q_1^0, F_1\rangle, \ldots, T_n \equiv \langle Q_n, \Sigma, \delta_n, q_n^0, F_n\rangle$
    $R := I$
    $A' := \bigwedge_{1 \le i \le n}$ unroll$(0, 1, T_i)$
    $B := \bigwedge_{1 \le i \le n}$ unroll$(1, k, T_i) \wedge F$
    while ($true$)
        $A := R \wedge A'$
        Run SAT solver on $A \wedge B$
        if $A \wedge B$ is satisfiable then
            if $R = I$ then
                return SAT
            else
                return INCONCLUSIVE
        else
            $P :=$ genInterpolant($A$,$B$)
            if $P[s_1/s_0] \models R$ then
                return UNSAT
            else
                $R := R \vee P[s_1/s_0]$
```

Fig. 4. Pseudo-code for the procedure based on unbounded model checking with interpolation for testing whether the intersection of multiple SFAs is empty

5.1 Finite Unrolling

We introduce a Boolean variable $\langle q_i^k \rangle$ to represent the automaton being in state q_i at time k, and $\langle e_{i,j}^k \rangle$ to represent the automaton transitioning from state q_i to q_j during the k^{th} step. We use $\psi_{i,j}^k$ to denote the corresponding transition constraint (we assume that all transitions between a pair of states are merged into a single edge). $\mathsf{pred}(q_j)$ denotes the set of states with an outgoing edge to q_j.

We can use these variables to encode the transition relation at each layer:

$$\bigwedge_{(q_i,\psi,q_j)\in\delta} (\neg\langle q_i^k\rangle \Rightarrow \neg\langle e_{i,j}^k\rangle) \wedge (\neg\langle\psi^k\rangle \Rightarrow \neg\langle e_{i,j}^k\rangle) \wedge \bigwedge_{(q_j\in Q)} (\bigwedge_{q_i\in\mathsf{pred}(q_j)} \neg\langle e_{i,j}^k\rangle) \Rightarrow \neg\langle q_j^{k+1}\rangle$$

$$\bigwedge_{(q_i,\psi,q_j)\in\delta} (\langle q_i^k\rangle \wedge \langle\psi^k\rangle \Rightarrow \langle e_{i,j}^k\rangle) \wedge \bigwedge_{(q_i,\psi,q_j)\in\delta} (\langle e_{i,j}^k\rangle \Rightarrow \langle q_j^{k+1}\rangle) \qquad (\star)$$

The formulas marked ($\star$) are not necessary for correctness but can reduce the state space of the problem.

However, directly encoding the final condition would require checking at *every* step whether every automaton is in an accept state. To avoid this, we allow the language accepted by each automaton to be padded with an additional termination character (denoted \$ in Fig. 5). We then only need to test for acceptance at the final step. Unlike a conventional automaton unrolling, where we unroll only from the start state, we must introduce all state variables at the top layer;

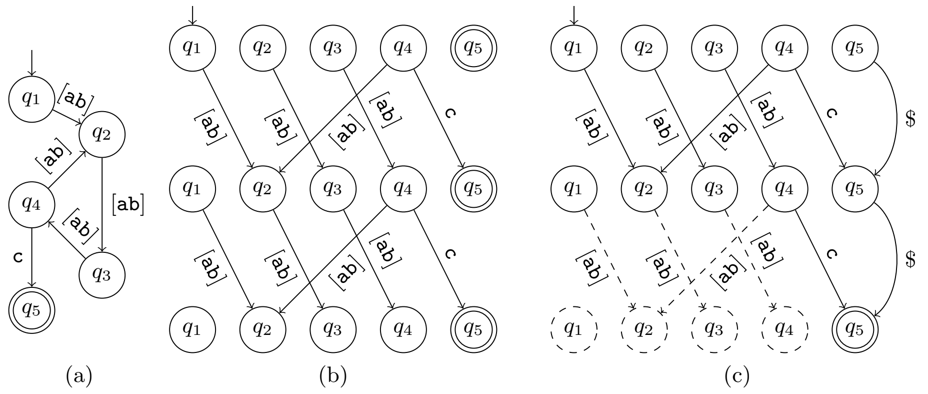


Fig. 5. Transition relation for an automaton of (a) ([ab]{3})+ c, (b) unrolled two steps, and (c) after adding transitions to allow for padding the end of string. States and edges that can be safely eliminated are shown dashed.

otherwise we cannot correctly compute the relaxed initial conditions, and may incorrectly conclude unsatisfiability.

In layers 2 to k, there may be states and edges which cannot reach an accept state in layer k. These states cannot affect the satisfiability of the overall clauses, and can be safely omitted.

Example 1. Consider the automaton shown in Fig. 5(a). The transition relation for this is (b) unrolled two steps, and then (c) corrected to allow for \$-padding. Consider the clauses generated for state q_5 in the second layer. We introduce $\langle e^1_{4,5}\rangle$ and $\langle e^1_{5,5}\rangle$ for the incoming edges, and $\langle q^1_5\rangle$ for the node, and the following formulae:

$$\begin{gathered}(\neg\langle q^1_4\rangle \Rightarrow \neg\langle e^1_{4,5}\rangle) \wedge (\neg\langle x_1 \in [\mathtt{ab}]\rangle \Rightarrow \neg\langle e^1_{4,5}\rangle)\\ (\neg\langle q^1_5\rangle \Rightarrow \neg\langle e^1_{5,5}\rangle) \wedge (\neg\langle x_1 = \$\rangle \Rightarrow \neg\langle e^1_{5,5}\rangle)\\ \neg\langle e^1_{4,5}\rangle \wedge \neg\langle e^1_{5,5}\rangle \Rightarrow \neg\langle q^2_5\rangle\end{gathered}$$

After generating similar clauses for each edge and node in the unrolled graph, we add the initial and final conditions requiring that the machine begins in the start state, and ends in an accept state:

$$I = \neg\langle q^0_2\rangle \wedge \neg\langle q^0_3\rangle \wedge \neg\langle q^0_4\rangle \wedge \neg\langle q^0_5\rangle \qquad\qquad F = \langle q^2_5\rangle$$

Notice the dotted states q^2_1 to q^2_4. The truth value of state q^2_5 is not dependent on the value of these states; as such, they cannot cause unsatisfiability, or affect the interpolant. In general, however, we require all variables for the first unrolled state in order to generate correct interpolants.

At the first iteration, this conjunction of formulas is clearly unsatisfiable; there is no path from q_1^0 to q_5^2. We then compute the interpolant for the system of constraints, yielding $P = \neg\langle q_4^1\rangle \wedge \neg\langle q_5^1\rangle$. This is not a fixed-point, since there is a solution satisfying P that doesn't satisfy I. At the second iteration, we compute the relaxed initial conditions $I' = I \vee P$ (which upon simplification gives P). As I' permits the machine to be in state q_3, the system of constraints is now satisfiable. So we cannot prove unsatisfiability at this depth; we must unroll the automaton further.

It may be tempting to also omit states which are known to be false, such as q_1^1 shown in Fig. 6. However, if $\langle q_1^1\rangle$ is omitted, a possible interpolant that may be generated is $P = \neg\langle q_2^1\rangle \wedge \neg\langle r_3^1\rangle$. When this is mapped back to the initial state, the algorithm will incorrectly detect satisfiability (with $q_1^0 \wedge r_2^0$), and unroll. The same interpolant will be generated after any number of unrolling steps, so the solver will never terminate.

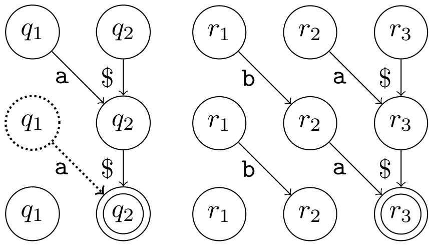


Fig. 6. Dotted state q_1^1 is always false, as it has no incoming edges. Still, it cannot be eliminated from the encoding.

5.2 Language Relaxation

Several of the languages tested in our first experiment in Section 6 generate automata with large numbers of states owing to the use of bounded repetition. The presence of these states can cause performance of the solver do degrade substantially; we conjecture that this is due to MathSAT not performing simplification of the generated interpolants, resulting in very large encodings of the set of reachable states.

However, in most of the cases we considered, the cause of unsatisfiability for the intersection of a pair of languages was unrelated to repetition operators. As a result, for regular expressions R_1 to R_n which make use of bounded repetition, we first check the satisfiability of $U(R_1) \cap \cdots \cap U(R_n)$, where U eliminates bounded repetition as follows:

$$\begin{array}{ll} U(e\{\mathtt{0},\mathtt{1}\}) = U(e)? & U(e_1 \ \mathbf{op}\ e_2) = U(e_1)\ \mathbf{op}\ U(e_2) \\ U(e\{\mathtt{0},\mathtt{j}\}) = U(e)* & U(\mathbf{op}(e)) = \mathbf{op}(U(e_1)) \\ U(e\{\mathtt{i},\mathtt{j}\}) = U(e)+ & \end{array}$$

If the intersection of these overapproximated languages is empty, we can terminate early without testing the full automata.

6 Experimental Results

To evaluate the method described in the previous sections, we have implemented a prototype solver, REVENANT,[1] in C++ using MathSAT5 [7] for SAT-solving and interpolant generation. All experiments have been run on a single core of a 2.7GHz Core i7-26202M with 7.8Gb memory. We compare the performance of REVENANT with REX [22][2] and DPRLE [9], and STRSOLVE [11][3] on a range of common benchmarks (first and second experiments).

Previous papers have focused on the intersection of only pairs of languages, for which the product automaton has in the worst case $O(n^2)$ states. However, a general solver for regular language constraints should be able to handle more complex systems of constraints. To test the performance of these methods on larger conjunctions of automata, we also present two classes of problems (third and fourth experiments) which exhibit more challenging behaviour.

Intersection of Multiple Languages. We generate intersections of multiple languages $\bigcap_{i \in \{2,\ldots,5\}} R_i$ such that R_i is each of the ten regular expressions extracted from some real-world applications that appeared originally in [15]. Table 1(a) shows the results of our evaluation running the different tools. Note that previous works (e.g., [22,9,11]) used the same set of regular expressions but regular set difference of pairs of languages was used instead of intersection. The reason why we do not perform the same experiment here is that our current implementation does not handle regular complement. Column T is the solving time of each tool, column T_{out} denotes the number of times that a timeout of 60 seconds expired, and S/U is the number of satisfiable versus unsatisfiable instances.

Unsurprisingly, REVENANT does not outperform the existing solvers in the case of pairs of automata, as it has the overhead of introducing $O(|R|k)$ variables and the corresponding clauses. However, as the number of languages increases, this up-front cost is outweighed by the gain from not generating the complete product automaton.

Generation of Long Strings. Our next experiment evaluates the performance of each solver for generating long strings from underconstrained systems. For this, we repeat an experiment from [22], probing the intersection of the regular expressions $[\mathtt{a}-\mathtt{c}]*\mathtt{a}[\mathtt{a}-\mathtt{c}]\{\mathtt{n}+\mathtt{1}\}$ and $[\mathtt{a}-\mathtt{c}]*\mathtt{b}[\mathtt{a}-\mathtt{c}]\{\mathtt{n}\}$. Table 1(b) shows, for various n, the time spent by each tool to generate a single string that matches both regular expressions. This is a worst-case scenario for our method since the two regular expressions are trivially satisfiable and therefore, our full encoding of the automata does not pay off.

[1] REVENANT is available at `http://ww2.cs.mu.oz.au/~ggange/revenant/`

[2] We run REX using the Mono framework 2.10.8.1.

[3] At the time of writing, the available STRSOLVE version [10] only supports intersection of pairs of languages.

Table 1. Comparison of REVENANT with existing string solvers, on several classes of regular expressions. All times are in seconds.

	REVENANT			REX			DPRLE			STRSOLVE		
	T	T_{out}	S/U	T	T_{out}	S/U	T	T_{out}	S/U	T	T_{out}	S/U
$i = 2$	4.48	0	22/23	38.78	0	22/23	2.08	0	22/23	0.32	0	22/23
$i = 3$	18.55	0	35/85	173.19	1	34/85	102.60	1	34/85	N/A	N/A	N/A
$i = 4$	130.88	1	35/174	401.22	4	31/175	613.71	7	28/175	N/A	N/A	N/A
$i = 5$	83.67	1	21/230	503.93	6	15/231	865.80	13	8/231	N/A	N/A	N/A

(a) Intersection of real-world regular expressions

	50	100	150	200	250	300	350	400	450	500
REVENANT	0.15	0.54	1.18	2.12	3.42	5.08	7.39	9.78	13.15	17.42
REX	0.10	0.16	0.27	0.46	0.73	1.24	1.92	2.90	4.00	5.54
DPRLE	0.01	0.06	0.09	0.17	0.25	0.36	0.48	0.65	0.78	0.96
StrSolve	0.00	0.00	0.02	0.03	0.04	0.06	0.09	0.11	0.17	0.21

(b) Generation of long strings

	2	4	6	8	10	12	14	16	18
REVENANT	0.01	0.02	0.04	0.06	0.06	0.05	0.09	0.08	0.14
REX	0.10	0.10	0.12	0.16	0.30	0.79	3.75	16.86	OutOfMemory
DPRLE	0.00	0.00	0.00	0.02	0.08	0.48	3.09	29.57	333.80

(c) Exponential branching

	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$
REVENANT	0.01	0.00	0.02	0.02	0.02	0.03
REX	0.10	0.10	0.18	3.27	OutOfMemory	OutOfMemory
DPRLE	0.00	0.00	0.03	0.40	15.21	OutOfMemory

(d) Exponential cycles

Exponential Branching. Even if we restrict attention to finite languages, the size of the product automaton may still be exponential in size. We construct a family of languages of the form

$$\begin{aligned} L_i = \quad & (\texttt{[0-1]\{i-1\}0[0-1]\{n-1\}0[0-1]\{n-i\}}\varphi_{\texttt{i}}) \\ & \mid (\texttt{[0-1]\{i-1\}1[0-1]\{n-1\}1[0-1]\{n-i\}}\varphi_{\texttt{i}}) \end{aligned}$$

such that $\varphi_1 \cap \cdots \cap \varphi_n$ is empty. An example language family of this kind is

$$\begin{aligned} L_1 &= \texttt{[0-1]\{0\}0[0-1]\{3\}0[0-1]\{3\}[bcd]} \mid \texttt{[0-1]\{0\}1[0-1]\{3\}1[0-1]\{3\}[bcd]} \\ L_2 &= \texttt{[0-1]\{1\}0[0-1]\{3\}0[0-1]\{2\}[acd]} \mid \texttt{[0-1]\{1\}1[0-1]\{3\}1[0-1]\{2\}[acd]} \\ L_3 &= \texttt{[0-1]\{2\}0[0-1]\{3\}0[0-1]\{1\}[abd]} \mid \texttt{[0-1]\{2\}1[0-1]\{3\}1[0-1]\{1\}[abd]} \\ L_4 &= \texttt{[0-1]\{3\}0[0-1]\{3\}0[0-1]\{0\}[abc]} \mid \texttt{[0-1]\{3\}1[0-1]\{3\}1[0-1]\{0\}[abc]} \end{aligned}$$

Table 1(c) shows the time for running the solvers for different values of n. For this experiment, we run REVENANT without relaxation, as the relaxed languages are trivially unsatisfiable. Clearly, this is an ideal case for REVENANT, as we can immediately prove unsatisfiability, where other solvers must explore the entire state space.

Exponential Paths. Conjunctions of languages may also contain cycles of exponential length. Consider the set of languages

$$L_1 = \texttt{[a-c]*([a-c]\{3\})+[bc]}$$
$$L_2 = \texttt{[a-c]*([a-c]\{5\})+[ac]}$$
$$L_3 = \texttt{[a-c]*([a-c]\{7\})+[ab]}$$

The intersection of languages $L_1 \cap L_2 \cap L_3$ is empty. However, as the cycle length of each language is coprime, both the product construction and search-based methods will generate all possible combinations of cycle-positions before the automata are synchronized at the loop exit, and the intersection can be proven empty. Table 1(d) shows the time spent for each tool to prove unsatisfiability. As in the previous case, we run REVENANT without relaxation. As before, REVENANT is substantially faster, as it can prove unsatisfiability without unrolling to the synchronization point.

The last two experiments have illustrated extreme cases in which REVENANT can significantly outperform the other existing tools. Similarly, we could construct other examples where our tool would perform very poorly. Consider the following set of languages similar to the previous one

$$L_1 = \texttt{[a-c]+[bc]d[a-c]\{3\}+}$$
$$L_2 = \texttt{[a-c]+[ac]d[a-c]\{5\}+}$$
$$L_3 = \texttt{[a-c]+[ab]d[a-c]\{7\}+}$$

In this case our SAT-based method, when used without relaxation, detects unsatisfiability due to the unsynchronized loop exits, rather than the $\varphi_{\texttt{i}}\texttt{d}$ chokepoint. The corresponding interpolant weakens the initial conditions too far, and the problem must be fully unrolled before unsatisfiability can be proven. With relaxation, however, we prove unsatisfiability without unrolling.

7 Conclusions and Further Work

We have described a new method for testing emptiness of the intersection of multiple regular languages, based on unbounded model-checking techniques. We have implemented a prototype solver, REVENANT, which uses this method; combined with language relaxation, REVENANT is competitive with existing solvers on realistic problem instances. We have also illustrated families of problems where this method is exponentially faster than existing techniques.

The differences between solvers on various families of constraints suggests that hybrid approaches should be studied, in particular for software verification. While our prototype currently handles only language intersection constraints, we intend to expand this to support concatenation constraints ($x \circ y \in L$), as well as negation and disjunction of constraints.

The relaxation described in Section 5.2 is an approximation of a traditional abstraction-refinement loop [4], where we only perform a single refinement step. It would be interesting to replace this with finer-grained progressive strengthening.

Our relaxation is currently a purely syntactic transformation. Such a scheme is only possible if the bounded repetition is already specified as part of the input. If the language is generated procedurally, or provided as an automaton, the transformation is no longer viable. Instead, it may be worthwhile to develop algorithms that examine the automaton directly for relaxation opportunities.

Finally, it would be interesting to apply the same technique to the testing of emptiness of context-free language intersection, by iteratively refining regular overapproximations to the languages involved. Chaki *et al.* [2] do this in a different context, namely the verification of concurrent C programs; an implementation was incorporated into the model checker MAGIC [3].

Acknowledgments. We wish to thank Pieter Hooimeijer for providing both software and valiant support on short notice. We acknowledge support of the Australian Research Council through Discovery Project Grant DP110102579.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing Context-Free Grammars Using an Incremental SAT Solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
2. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying Concurrent Message-Passing C Programs with Recursive Calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
3. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Transactions on Software Engineering 30(6), 388–402 (2004)
4. 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)
5. Craig, W.: Linear reasoning: A new form of the Herbrand-Gentzen theorem. Journal of Symbolic Logic 22(3), 250–268 (1957)
6. 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)
7. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8, 1–27 (2012)
8. Hooimeijer, P., Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011)
9. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Proc. 2009 ACM SIGPLAN Conf. Programming Language Design and Implementation, pp. 188–198. ACM (2009)

10. Hooimeijer, P., Weimer, W.: Solving string constraints lazily. In: Proc. IEEE/ACM Conf. Automated Software Engineering, pp. 377–386 (2010)
11. Hooimeijer, P., Weimer, W.: StrSolve: Solving string constraints lazily. Automated Software Engineering 19(4), 531–559 (2012)
12. Ilie, L., Solis-Oba, R., Yu, S.: Reducing the Size of NFAs by Using Equivalences and Preorders. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 310–321. Springer, Heidelberg (2005)
13. Ilie, L., Yu, S.: Reducing NFAs by invariant equivalences. Theoretical Computer Science 306(1-3), 373–390 (2003)
14. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for string constraints. In: Proc. 18th Int. Symp. Software Testing and Analysis (ISSTA 2009), pp. 105–116. ACM (2009)
15. Li, N., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Reggae: Automated test generation for programs using complex regular expressions. In: Proc. 24th IEEE/ACM Int. Conf. Automated Software Engineering, pp. 515–519 (2009)
16. 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)
17. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science 345(1), 101–121 (2005)
18. Minamide, Y.: Static approximation of dynamically generated web pages. In: Proc. 14th Int. Conf. World Wide Web, pp. 432–441. ACM Press (2005)
19. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(2), 981–998 (1997)
20. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proc. IEEE Symp. Security and Privacy, pp. 513–528. IEEE Computer Society (2010)
21. Veanes, M., de Halleux, P., Tillman, N.: Rex: Symbolic regular expression explorer. Microsoft Research Technical Report MSR-TR-2009-137, Microsoft Research, Redmond, WA (2009)
22. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Proc. Third Int. Conf. Software Testing, Verification and Validation, pp. 498–507. IEEE Comp. Soc. (2010)
23. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proc. ACM SIGPLAN 2007 Conf. Programming Language Design and Implementation, pp. 32–41 (2007)

eVolCheck: Incremental Upgrade Checker for C*

Grigory Fedyukovich[1], Ondrej Sery[1,2], and Natasha Sharygina[1]

[1] University of Lugano, Switzerland
{name.surname}@usi.ch
[2] D3S, Faculty of Mathematics and Physics, Charles University, Czech Rep.

Abstract. Software is not created at once. Rather, it grows incrementally version by version and evolves long after being first released. To be practical for software developers, the software verification tools should be able to cope with changes. In this paper, we present a tool, eVolCheck, that focuses on incremental verification of software as it evolves. During the software evolution the tool maintains abstractions of program functions, function summaries, derived using Craig interpolation. In each check, the function summaries are used to localize verification of an upgrade to analysis of the modified functions. Experimental evaluation on a range of various benchmarks shows substantial speedup of incremental upgrade checking of eVolCheck in contrast to checking each version from scratch.

1 Introduction

Software is rarely stable. Not only it gradually evolves during its development, but it is also subject to changes after it is released (e.g., bug fixes, component upgrades, platform changes, etc.). This evolution is an inherent part of software development and as such, it should be reflected also by the software verification tools. With this in mind, we developed a tool called eVolCheck, which focuses on incremental verification of software written in C.

The eVolCheck tool is a bounded model checker (BMC), which was specifically designed to handle incremental changes by focusing on the actual changes and to avoid resorting to the re-verification of the updated systems from scratch as most tools have to do in the presence of changes. In particular, it uses interpolation-based function summaries to localize and thus speedup the checks of new versions of a software. Concretely, eVolCheck maintains over-approximating summaries of all the program functions. After a change, it first attempts to verify that the old summaries are still valid for the changed program functions. Since this check considers only code of the function bodies, its old summary and potentially summaries of its callees, it is very local and thus it tends to be computationally inexpensive. If it succeeds, the upgrade is safe. Otherwise, the check is propagated to the callers of the modified functions. When the summary of the call tree root is shown to be violated, a real error is found and it is reported to the user along with an error trace. After each successful check, any invalidated summaries are regenerated so that they are ready for the check of the next version.

* This work is partially supported by the European Community under the call FP7-ICT-2009-5 — project PINCETTE 257647.

In addition, `eVolCheck` features a counter-example guided refinement to deal with too coarse summaries during the checks.

The upgrade checking algorithm was originally described in [18] along with a discussion on its correctness. This paper focuses on the actual implementation of the `eVolCheck` tool, including an `Eclipse` plug-in, which facilitates its use, together with details of its industrial and academic applications.

The paper is structured as follows. In Section 2, we review the theoretical background of the algorithm with references for more detailed explanation. Section 3 describes the architecture of the `eVolCheck` tool together with the essential implementation details, while Section 4 focuses on the usage of the tool and its integration into `Eclipse`. In Section 5 we present experimental evaluation on various benchmarks. We list the related work in Section 6 and conclude in Section 7.

2 Background Theory

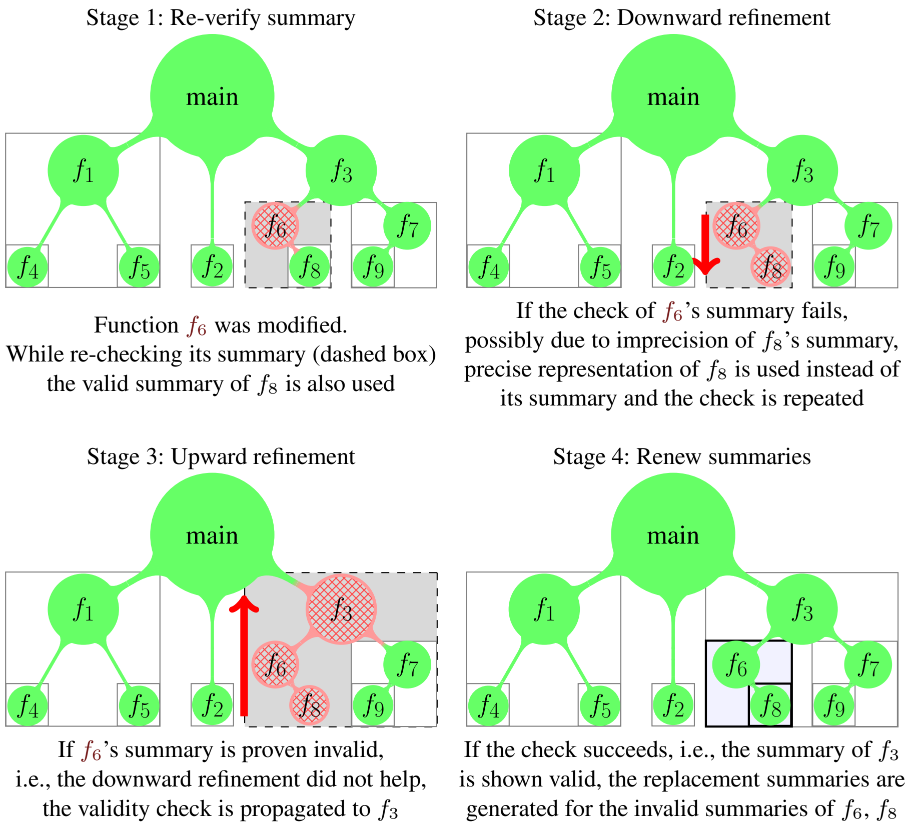
This section focuses on the theoretical background of the upgrade checking algorithm which is core of `eVolCheck`.

Upgrade Checking. During the software evolution, `eVolCheck` maintains over-approximations of input/output behaviors of all functions in the code, i.e., function summaries. Initially, the function summaries are generated during a bootstrapping run, which is equivalent to the standalone verification and implements the approach of [16]. Then, the function summaries are validated (some potentially replaced) during each successful upgrade check.

In each check, `eVolCheck` first identifies the set of modified functions on the syntactical level. For this purpose, we developed a tool called `goto-diff` which effectively detects the semantic changes (more details are in Section 3). Then `eVolCheck` attempts to show that the old function summaries are still valid over-approximations of the behavior of the modified functions (Stage 1 in Fig. 1). These are local and thus cheap checks. As an option of `eVolCheck`, to further speed up the check, all the valid summaries may be used in this check to abstract the corresponding function calls.

If the local checks succeed, the upgraded version is safe. If not, it can be either because the valid summaries of the called functions are not precise enough, in which case they are replaced by their precise representation, by performing *downward refinement* (Stage 2 in Fig. 1). Or it can be because the summary is indeed violated during the change, in which case the check is propagated to the parent functions, by performing *upward refinement* (Stage 3 in Fig. 1). This is iterated until either the check succeeds (Stage 4 in Fig. 1) on some level of the call tree, or the check fails for the `main` function in the root of the call tree. In the former case, new valid summaries are generated for the subtree, while in the latter case, a real error is identified and reported to the user.

As a result `eVolCheck` exploits the locality of the changes, which makes it a valuable tool for efficient verification of fine-grained changes that have limited impact throughout the code. Of course, should the change be extensive and span the entire code base, naturally, the check can become expensive. This is in line with the envisioned position of the tool in the development process to check changes on the level of individual commits rather than major revisions.


Stage 1: Re-verify summary
Stage 2: Downward refinement
Stage 3: Upward refinement
Stage 4: Renew summaries
main
f_1
f_2
f_3
f_4
f_5
f_6
f_7
f_8
f_9

Function f_6 was modified.
While re-checking its summary (dashed box)
the valid summary of f_8 is also used

If the check of f_6's summary fails,
possibly due to imprecision of f_8's summary,
precise representation of f_8 is used instead of
its summary and the check is repeated

If f_6's summary is proven invalid,
i.e., the downward refinement did not help,
the validity check is propagated to f_3

If the check succeeds, i.e., the summary of f_3
is shown valid, the replacement summaries are
generated for the invalid summaries of f_6, f_8

Fig. 1. eVolCheck principle

Interpolation. The upgrade checking algorithm is based on over-approximating function summaries, however, it is not strictly tied to any particular form of abstraction or means to derive the summaries. Of course, the particular summaries need to satisfy certain properties, e.g., the correctness Properties 1 and 2 (formally defined later in this Section), used to show correctness of the algorithm. Our implementation of the algorithm in `eVolCheck` uses function summaries derived by Craig interpolation [5]. In a nutshell, given two formulas A and B such that $A \wedge B$ is unsatisfiable, a Craig interpolant of A and B is a formula I, s.t., $A \implies I$, and $I \wedge B$ is unsatisfiable, and I contains only the shared free variables of A and B.

Intuitively, interpolants are an over-approximation of formula A still capturing the conflict with B, while using only the shared language of (A, B). Craig interpolants are usually constructed from a resolution proof of unsatisfiability of $A \wedge B$ and they have numerous applications in model checking (see, e.g., [11]). Note that interpolants of different strength (considering implication relation) can be obtained using different interpolation algorithms. In practice, additional properties of multiple interpolants generated from a single unsatisfiable formula are often required, resulting in *path interpolants* and *tree interpolants*. Note that it is often possible to ensure these additional properties by careful construction of interpolants from the same proof of unsatisfiability [15].

Definition 1. *Let $A \wedge B \wedge C$ be an unsatisfiable formula and I_A, I_B, I_{AB} be Craig interpolants of $(A, B \wedge C)$, $(B, A \wedge C)$, and $(A \wedge B, C)$ respectively. The interpolants I_A, I_B, I_{AB} have the* tree interpolant *property iff $I_A \wedge I_B \implies I_{AB}$.*

In the implementation of `eVolCheck` algorithms, the tree interpolant property is essential as it must be satisfied to maintain valid function summaries and to ensure the correctness of the overall local upgrade checking.

Function Summarization. Standard BMC creates a monolithic formula not well suited for interpolation, as symbols of different scopes get mixed in the formula both due to the encoding and optimizations. To solve this problem, we create a so called *partitioned bounded model checking formula* (PBMC formula) that isolates variables of functions in separate conjuncts of the formula and shares only the interface symbols of functions. That is input and output parameters[1] and a few helper symbols, as further explained in [16,17]. In particular, for each function call f, there is a helper propositional variable error_f, that evaluates to true when an error (assertion violation) is reachable in that function given the valuation of its input parameters.

When the PBMC formula is unsatisfiable, it is easy to partition it for interpolant generation for each function call, so that A corresponds to the function implementation (including its callees) and B to the calling context. The generated interpolants are then over-approximations of the functions input/output behavior and contain only the interface variables of the functions. In other words, the interpolants constitute over-approximating function summaries.

Correctness of Upgrade Checking. The correctness of the local upgrade checking algorithm is based on maintaining the following two properties:

$$\mathit{error}_{f_{main}} \wedge \sigma_{f_{main}} \rightarrow \bot \tag{1}$$

Given each function call f and its children calls $g_1, \ldots, g_n$:

$$\sigma_{g_1} \wedge \ldots \wedge \sigma_{g_n} \wedge \phi_f \rightarrow \sigma_f \tag{2}$$

Property 1 claims that the entire program is safe by the means of the summary of the `main` function, $\sigma_{f_{main}}$, and its inconsistency with the $\mathit{error}_{f_{main}}$ capturing reachability of an error in the call tree of `main`, i.e., the entire program.

Property 2 requires that the summaries of callees (σ_{g_i}) along with precise representation of the body of the caller (ϕ_f) are captured by the summary of the caller (σ_f). In other words, that the over-approximations of the callees are not too weak to be captured by the over-approximation of the caller.

With these two properties, correctness of the upgrade checking algorithm is easy to see. It suffices to recursively apply Property 2 to replace summaries occurring in Property 1. The results state that the precisely encoded program is error free.

[1] Note that accessed global variables are handled as additional input/output arguments.

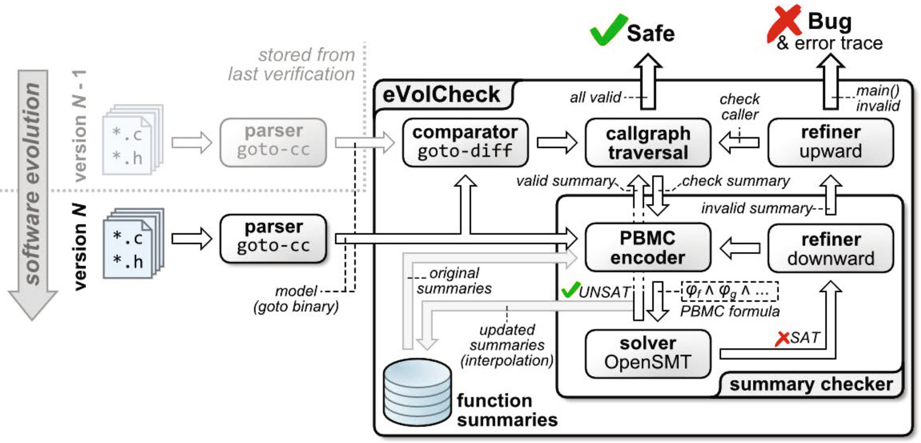


Fig. 2. eVolCheck architecture overview

In [18], we showed that the properties are established during the initial bootstrapping run, when all the summaries are generated, and that they are reestablished after each successful run of the upgrade checking algorithm. The proof relies on the tree interpolant property. This becomes transparent when the inductive nature of both Property 2 and Def. 1 is observed side by side.

3 Tool Architecture

This section presents the architecture of the eVolCheck tool as depicted in Fig. 2. The tool uses the goto-cc compiler provided by the CProver framework[2]. The goto-cc compiler produces an input model of the source code of C program (called *goto-binary*) suitable for automated analysis. Each version of the analyzed software is compiled using goto-cc separately. The resulting models are stored for future checks.

eVolCheck. The eVolCheck tool itself consists of a comparator, a call graph traversal, an upward refiner and a summary checker. The comparator identifies the changed functions calls. Note that if a function call was newly introduced or removed (i.e., the structure of the call graph is changed), it is considered as change in the parent function call. The call graph traversal attempts to check summaries of all the modified function calls bottom up. The upward refiner identifies the parent function call to be rechecked when a summary check fails. The summary checker performs the actual check of a function call against its summary. In turn, it consists of a PBMC encoder that takes care of unwinding loops and recursion, generation of SSA form and bit-blasting, a solver wrapper that takes care of communication with the solver/interpolator (OpenSMT [2]), and a downward refiner that identifies ancestor functions to be refined when a summary check fails possibly due to imprecise representation of the ancestor function calls. Additionally, there are two optional optimizations in eVolCheck, namely slicing and

[2] www.cprover.org

summary optimization. The first can reduce the size of the SSA form using slicing w.r.t. variables, irrelevant to the properties being checked. The second can compare the existent summaries for the same function and the same bound, and keep the more precise one.

Goto-diff. For comparing the two models, of the previous and the newly upgraded versions, we implemented a tool called `goto-diff`. The tool accepts two goto-binary models and analyzes them function by function. the longest common sub-sequence algorithm is used to match the preserved instructions and to identify the changed ones.

It is crucial that `goto-diff` works on the level of the models rather then on the level of the source files. This way, it is able to distinguish some of the inconsequential changes in the code. Examples include changes in the order of function declarations and definitions, text changes in comments and white spaces, and simpler cases of refactoring. These changes are usually reported as semantic changes by the purely syntactic comparators (e.g., the standard diff tool). Moreover, as `goto-diff` works on the goto-binary models (i.e., after the C pre-processors) it correctly interprets also changes in the pre-processor macros.

Solver and Interpolation Engine. As mentioned in Section 2, to guarantee correctness of the upgrade check, `eVolCheck` requires a solver that is able to generate multiple interpolants with the tree interpolant property from a single satisfiability query. For this reason, we use the interpolating solver, `OpenSMT`, which creates multiple interpolants from the same unsatisfiability proof and provides API for convenient specification of the partitions corresponding to the functions in the call tree. Currently, we use `OpenSMT` in the SAT solving mode and bit-blast all formulas to the propositional level. As a result, `eVolCheck` provides bit-precise reasoning.

Eclipse Plug-in. In order to make the tool as user-friendly as possible, we integrated `eVolCheck` in the `Eclipse` development environment in the form of a plug-in. For a user, developing a program using the `Eclipse` environment, the `eVolCheck` plug-in makes it possible to verify changes as part of the development flow for each version of the code. If the version history of the program is empty, the bootstrapping (initial verification) is performed first. Otherwise, `eVolCheck` verifies the program with respect to the last safe version. Graphical capabilities of `Eclipse` contain a variety of helpers, allowing configuration of the verification environment.

The plug-in is developed using Plug-in Development Environment (PDE), a toolset to create, develop, test, debug, build and deploy `Eclipse` plug-ins. It is built as an external jar-file, which is loaded together with `Eclipse`. The plug-in follows the paradigm of Debugging components, and provides the separate perspective, containing a view of the source code, highlighted lines, reported by `goto-diff`, visualization of the error traces and change impact, computed for each upgrade checking of the program.

At the low level, the plug-in delegates the verification tasks to the corresponding command line tools `goto-cc`, `goto-diff` and `eVolCheck`. It maintains a database and external file storage to keep goto-binaries, summaries and other meta-data of each version of each program verified earlier.

4 Tool Usage

The eVolCheck can be run from a command line as well as using the Eclipse plug-in. Its Linux binaries, benchmarks used for evaluation, a tutorial explaining how to use eVolCheck and explanation of the most important parameters are available on-line for other researchers[3].

The following shows the example of usage of eVolCheck from a command line [4]:

1. Create a model of the base version of the program by running the goto-cc compiler. Choose one of the *_orig.c files in examples directory and type:

```
~/evolcheck$ ./goto-cc examples/valid/change_valid_orig.c
     -o examples/valid/change_valid_orig.out
```

The file examples/valid/change_valid_orig.c is the input source code and examples/valid/change_valid_orig.out is the resulting goto-binary. Note that the upgrade checking environment should be prepared for analysis by cleaning the repository with `~/evolcheck$ rm __summaries __omega` before performing this step.

2. Run eVolCheck to perform the initial bootstrapping check of the program (parameter --init-upgrade-check):

```
~/evolcheck$ ./evolcheck --init-upgrade-check --unwind 10
     examples/valid/change_valid_orig.out
```

Note that the parameter --unwind <N> is required to specify the maximal number of unwindings of each loop.

3. Check the eVolCheck outputs. The following message at the end of the eVolCheck output indicates either that the program is safe:

```
ASSERTION(S) HOLD(S).
```

or that the program is buggy:

```
ASSERTION(S) DO(ES)N'T HOLD.
A real bug found.
```

In the latter case, a corresponding error trace manifesting the bug is part of the output as well. After a successful bootstrapping check, the summaries and their mapping to the calltree are created and stored for the subsequent upgrade checks in files __summaries and __omega respectively.

4. When the program is upgraded, goto-binary model of the new version is created again using the goto-cc compiler. Run it for the file corresponding *_upgr.c file chosen in Step 2:

```
~/evolcheck$ ./goto-cc examples/valid/change_valid_upgr.c
     -o examples/valid/change_valid_upgr.out
```

[3] http://www.verify.inf.usi.ch/evolcheck.html

[4] The running example can be found at http://www.inf.usi.ch/phd/fedyukovich/evolcheck.lin32.tar.gz

5. With the goto-binary model of the new version of the program, the actual upgrade check is performed (parameter `--do-upgrade-check <file>`) as follows:

```
~evolcheck$ ./evolcheck --do-upgrade-check
    examples/valid/change_valid_upgr.out
    --unwind 10 examples/valid/change_valid_orig.out
```

Note that the parameter `--unwind <N>` is required to specify the same unwinding number as for the original check.

6. Check the `eVolCheck` output. There are several possible cases. Either the two programs have identical models, i.e., no or only simple syntactical changes occurred (examples for this case are located in the `examples/ident` folder), resulting in the following output: `The program models are identical.`

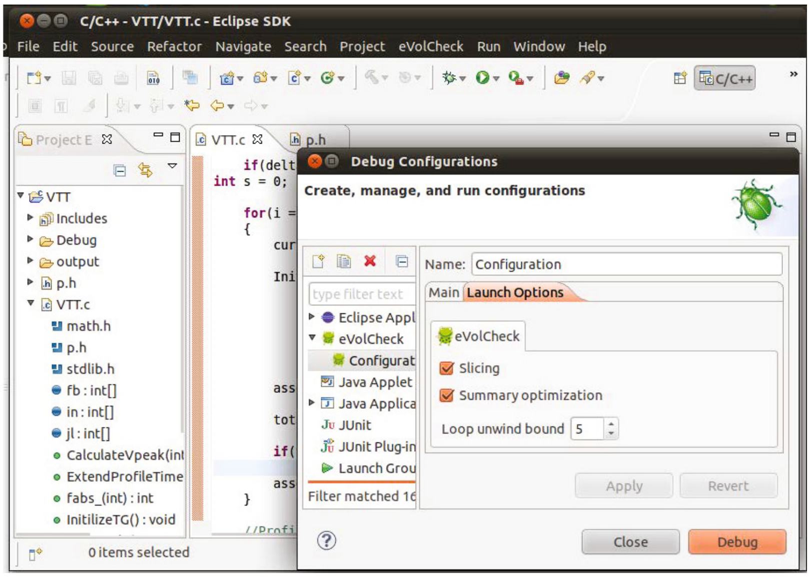
Or the upgraded program was changed but it remains correct (examples are located in the `examples/valid` folder), resulting in the following message for each checked function summary: `... summary was verified.`

Or the upgraded program is buggy (examples from `examples/not_valid`). The corresponding output contains the following message for the summary of the function `main`:

```
Old summary is no more valid.
...
summary cannot be renewed. A real bug found.
```

7. Additional information about the usage of the tool can be found simply by typing

```
~/evolcheck$ ./evolcheck --help
```



Name: Configuration

eVolCheck

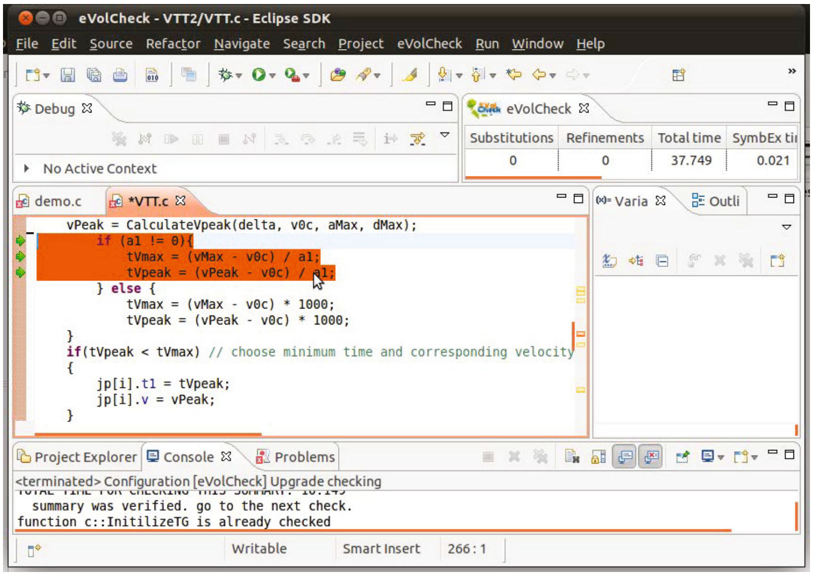
☑ Slicing

☑ Summary optimization

Loop unwind bound 5

Fig. 3. eVolCheck configuration window

Eclipse Plug-in. Nowadays, IDEs form an essential part of software development tool chains. Therefore, we integrated `eVolCheck` into `Eclipse`, which is one of the most widely used IDE. Our plug-in hides some of the implementation details and provides

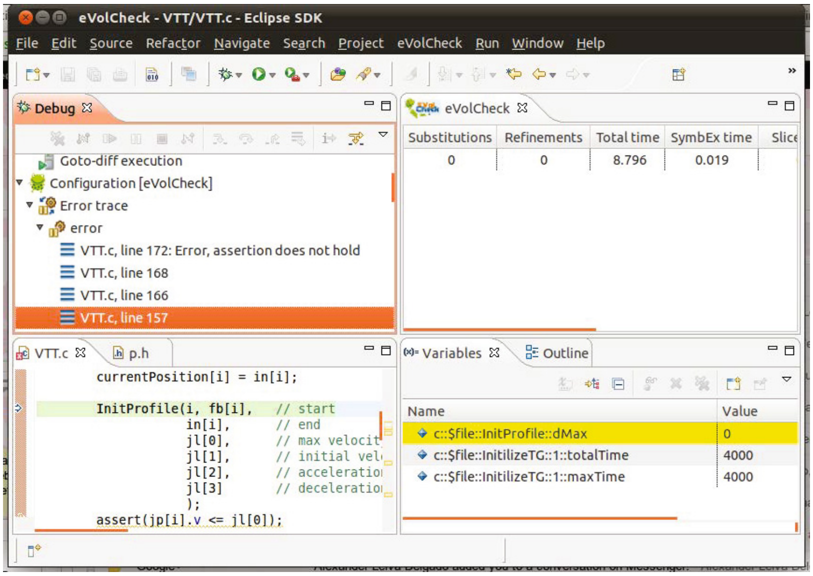

eVolCheck - VTT2/VTT.c - Eclipse SDK
File Edit Source Refactor Navigate Search Project eVolCheck Run Window Help
Debug
No Active Context
eVolCheck
demo.c
*VTT.c
Varia
Outli
Project Explorer
Console
Problems
Writable
Smart Insert
266 : 1

Substitutions	Refinements	Total time	SymbEx ti
0	0	37.749	0.021

```
    vPeak = CalculateVpeak(delta, v0c, aMax, dMax);
        if (a1 != 0){
            tVmax = (vMax - v0c) / a1;
            tVpeak = (vPeak - v0c) / a1;
        } else {
            tVmax = (vMax - v0c) * 1000;
            tVpeak = (vPeak - v0c) * 1000;
    }
    if(tVpeak < tVmax) // choose minimum time and corresponding velocity
    {
        jp[i].t1 = tVpeak;
        jp[i].v = vPeak;
    }
```

```
<terminated> Configuration [eVolCheck] Upgrade checking
TOTAL TIME FOR CHECKING THIS SUMMARY: 10.149
  summary was verified. go to the next check.
function c::InitilizeTG is already checked
```

Fig. 4. eVolCheck invokes goto-diff (changed lines are highlighted)


eVolCheck - VTT/VTT.c - Eclipse SDK
File Edit Source Refactor Navigate Search Project eVolCheck Run Window Help
Debug
eVolCheck
VTT.c
p.h
Variables
Outline

```
Goto-diff execution
▾ Configuration [eVolCheck]
  ▾ Error trace
    ▾ error
        VTT.c, line 172: Error, assertion does not hold
        VTT.c, line 168
        VTT.c, line 166
        VTT.c, line 157
```

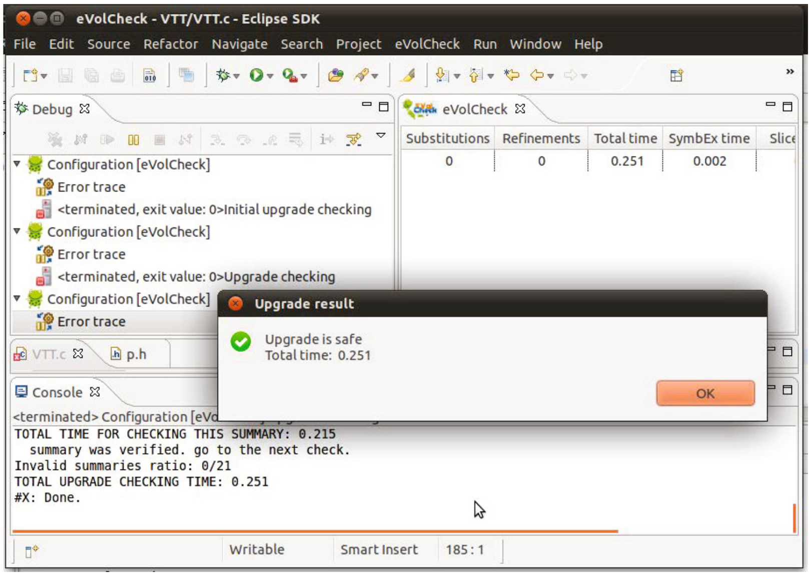
Substitutions	Refinements	Total time	SymbEx time	Slice
0	0	8.796	0.019	

```
        currentPosition[i] = in[i];

        InitProfile(i, fb[i],    // start
                    in[i],       // end
                    jl[0],       // max velocit
                    jl[1],       // initial vel
                    jl[2],       // acceleratio
                    jl[3]        // deceleratio
                    );
        assert(jp[i].v <= jl[0]);
```

Name	Value
c::$file::InitProfile::dMax	0
c::$file::InitilizeTG::1::totalTime	4000
c::$file::InitilizeTG::1::maxTime	4000

Fig. 5. eVolCheck error trace


eVolCheck - VTT/VTT.c - Eclipse SDK
File Edit Source Refactor Navigate Search Project eVolCheck Run Window Help
Debug
Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Initial upgrade checking
Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Upgrade checking
Configuration [eVolCheck]
Error trace
VTT.c
p.h
Console
eVolCheck
Upgrade result
Upgrade is safe
Total time: 0.251
OK
Writable
Smart Insert
185 : 1

Substitutions	Refinements	Total time	SymbEx time	Slice
0	0	0.251	0.002	

```
<terminated> Configuration [eVo
TOTAL TIME FOR CHECKING THIS SUMMARY: 0.215
  summary was verified. go to the next check.
Invalid summaries ratio: 0/21
TOTAL UPGRADE CHECKING TIME: 0.251
#X: Done.
```

Fig. 6. eVolCheck successful verification report

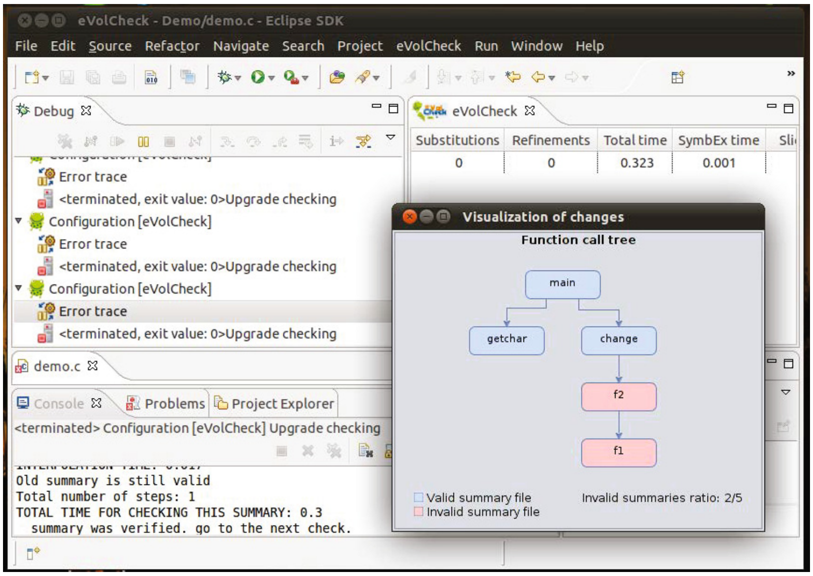
much more comfort compared to the command line tool. As expected, the actual use of the plug-in follows the command line scenario.

1. The user develops a current version of the program. In order to specify properties, the assertions should be placed in the code or generated automatically by the tool. The examples of the default properties are division by zero, pointers dereferencing, array out-of-bounds checks.

2. The user opens the *Debug Configurations* window and chooses the file(s) to be checked and specifies the unwinding bound (Fig. 3). `Eclipse` then automatically creates the model (goto-binary) from the selected source files and keeps working with it.

3. The plug-in searches for the last safe version of the current program (goto-binary created from the same selection of source files and the same unwinding number). If no such a version is found, it performs the initial bootstrapping check. Otherwise, plug-in restores the summaries and outdated goto-binary from the subsidiary storage. `eVolCheck` then identifies the modified code by comparing call trees for both the current and the previous versions. The modified lines of code are marked (Fig. 4) for the user review. Note that modified code may also contain some new properties, manually or automatically inserted. These properties will be also considered in the next step.

4. Then the localized upgrade check is performed. If it is unsuccessful, the plug-in reports violation to the user and provides an error trace (Fig. 5). The user can traverse the error trace line by line in the original code and see the valuation of all variables in all states along the error-trace. If desired, the user fixes the reported errors and continues from Step 3.



Error trace
<terminated, exit value: 0>Upgrade checking
Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Upgrade checking
Configuration [eVolCheck]
Error trace
<terminated, exit value: 0>Upgrade checking

Substitutions	Refinements	Total time	SymbEx time	Sli
0	0	0.323	0.001	

<terminated> Configuration [eVolCheck] Upgrade checking

```
Old summary is still valid
Total number of steps: 1
TOTAL TIME FOR CHECKING THIS SUMMARY: 0.3
  summary was verified. go to the next check.
```

☐ Valid summary file
🟥 Invalid summary file
Invalid summaries ratio: 2/5

Fig. 7. eVolCheck change impact

5. In case of successful verification, the positive result is reported (Fig. 6). The plug-in stores the set of valid and new summaries and the goto-binary in the subsidiary storage. In addition, graphical visualization of the change impact in the form of a colored call-tree is available (Fig. 7).

5 Evaluation

In addition to the standalone use of `eVolCheck` (as described in Section 4), the tool is used as a static analysis engine within the hybrid static/dynamic upgrade checking platform developed as part of the Pincette project[5]. The platform has been applied to analysis of software developed by Pincette's industrial partners, among which there are the VTT company with its control software for a maintenance robot for the ITER fusion reactor; the IAI company with the software for a stabilized optical device payload (MSEOS) of their unmanned airborne vehicles; and the ABB company with the software of their power grid protection units. As part of the project, `eVolCheck` is also integrated in the CCRT platform[6], a collaborative code review tool developed at IBM.

The `eVolCheck` tool was validated on a wide-range of various benchmarks among which are the validation cases, provided by the Pincette project collaborators. In particular, it was used to verify the C part of the implementation of the DTP2 robot controller, developed by the VTT company. It was also applied to the ABB validation cases on a code taken from the project implementing a core of a feeder protector and controller.

[5] `http://www.pincette-project.eu`
[6] CCRT is a proprietary tool of IBM.

Table 1. Experimental evaluation

Benchmark	Bootstrap		Upgrade check					
Name	Total [s]	Itp [s]	Total [s]	Diff [s]	Itp [s]	**Speedup**	Result	ISR
`ABB_A`	8.644	0.008	0.04	0.009	0.003	**220**x	SAFE	0/7
`ABB_B`	6.236	0.009	0.006	0.006	—	**935**x	SAFE	0/9
`ABB_C`	8.532	0.015	0.059	0.008	0.003	**157**x	SAFE	0/8
`VTT_A`	0.512	0.001	0.006	0.006	—	**85.5**x	SAFE	0/9
`VTT_B`	0.514	0.001	0.031	0.006	—	**0.7**x	BUG	1/9
`euler_A`	12.56	0.099	0.179	0.001	0.016	**70.4**x	SAFE	1/6
`euler_B`	12.547	0.095	2.622	0.001	0.031	**4.74**x	SAFE	3/5
`life_A`	13.911	1.366	0.181	0.001	<0.001	**77.0**x	SAFE	0/5
`life_B`	13.891	1.357	6.774	0.001	—	**0.31**x	BUG	5/5
`arithm_A`	0.147	0.007	0.355	0.001	—	**0.39**x	BUG	3/3
`diskperf_A`	0.167	0.001	0.024	0.008	<0.001	**5.79**x	SAFE	0/21
`diskperf_B`	0.137	0.001	0.062	0.009	—	**2.25**x	BUG	3/21
`floppy_A`	2.146	0.229	0.422	0.202	<0.001	**5.02**x	SAFE	0/226
`floppy_B`	2.183	0.237	2.277	0.206	—	**0.82**x	BUG	79/226
`kbfiltr_A`	0.288	0.011	0.081	0.023	0.001	**3.40**x	SAFE	1/63
`kbfiltr_B`	0.320	0.009	0.088	0.023	0.001	**1.85**x	SAFE	3/63

The code originates from an embedded software used in the ABB hardware module. This is a large scale project containing many sub-projects which implement various functions of the feeder device. The total number of lines in the overall code is in millions. Pre-processing the code with the `goto-cc` tool generated a collection of goto-binaries (each one represents a separate source file, estimated by thousands lines of code) that were then processed with `eVolCheck` focusing the validation to particular functional sub-projects.

To demonstrate the applicability and advantages of `eVolCheck`, we provide evaluation details of several test cases. Five of them (`ABB_`n, `VTT_`n) were provided by the Pincette project partners for which the changes were extracted from the project repositories. Six other benchmarks were derived from Windows device driver library (`diskperf_`n, `floppy_`n, `kbfiltr_`n). The changes (with different level of impact, from adding an irrelevant line of code to moving a part of functionality between functions) were introduced manually there. Roughly, all benchmarks are hundreds to thousands lines of code each. The rest of the benchmarks are the masters' student projects conducted at University of Lugano.

Table 1 represents results of the experiments. Each benchmark is shown in a separate row, which summarizes statistics about the initial verification and verification of an upgrade. Time (in seconds) for running the syntactic difference check (**Diff**) and for generation of the interpolants (**Itp**) represents the computational overhead of the upgrade checking procedure, and included in the total running time (**Total**) of `eVolCheck`. Note that interpolation can not be performed at the buggy examples (marked as "—"), for which the corresponded PBMC formula is satisfiable. To show advantages of our upgrade checking approach, for each change we calculated the speedup (**Speedup**) of the upgrade check versus standalone verification of the changed code from scratch, performed only for the sake of comparison and thus not shown in the table. Finally, the posteriori estimation of the upgrade check complexity is shown in the row **ISR** (Invalid Summaries Ratio). This ratio represents the number of invalid summaries (due to the change) with respect to the number of nodes in the call tree of the verified program.

Discussion. Our evaluation demonstrates good performance of `eVolCheck`. In particular, the experiments show high efficiency of upgrade checking for safe upgrades since they result in a small number of refinements (both, upward and downward). This generally leads to a small number of invalidated summaries, as witnessed by the corresponding **ISR** (see, for example, the ABB_n cases, where summaries of all changed functions were proven valid). It is less efficient (for some tests) in case of buggy upgrades, since bugs frequently (as expected) effect larger portions of the program. In classical model checking, confirming the absence of bugs is usually more expensive (since it requires the full state-space search) then detecting the bugs (where the search can be terminated once the bug is detected) and we believe that the fact that `eVolCheck` works so well to confirm safety is very useful for routine analysis of upgrades.

The use of `goto-diff` has been very useful since it managed to detected many test cases with small syntactic changes which did not require running the main `eVolCheck` procedures. For example, in `VTT_A` and `ABB_B`, the comparator proved that the models are identical, so no further checking was needed.

As expected, in the majority of the experiments, the localized upgrade check provided by `eVolCheck` outperforms the verification from scratch, which is indicated by **speedup** > 1. Moreover, in many instances (usually on large industrial cases) the speedup is large, which demonstrates good efficiency and usefulness of the tool.

6 Related Work

The general idea of interpolation-based function summarization was studied in various projects including earlier work of `eVolCheck` authors [12,13,1,16,8]. For instance, the authors of [8] generate sequence of inductive interpolants as summaries of recursive functions to be used in Hoare-style verification of a single (not upgraded) program. To our knowledge, there is no implementation and upgrade checking is not considered. A tool called `FunFrog` [17] is an implementation of the idea from [16]. To the best of our knowledge, `eVolCheck` is the first tool which further extends interpolation-based function summaries to incremental checking of software upgrades.

Approaches [7,19] also employ bottom-up call graph traversal. In [19], the authors start from possible error location in order to prove its unreachability in a context of a current function. If not proven, they expand the context to the caller functions and repeat the check. They have an implementation for Java programs, but do not consider an upgrade checking case.

Previously, other researchers attempted to reuse (parts of) models constructed during verification of the base version to speed up verification of software upgrades. Either, the models constituted an entire reachable abstract state space [9,4] that was revalidated after a change. Alternatively, behavioral models of different components of the software [3] were constructed (by employing techniques for learning regular languages) and then substitutability between the original and the altered components was analyzed after each change. In comparison, eVolCheck stores information corresponding to the function calls. We argue that this is a natural abstraction boundary that is more stable than the abstract state space and at the same time more fine-grained than the entire software components.

There are also approaches that attempt to show equivalence of the original and the upgraded software [14,10,7]. The SymDiff tool [10] decides conditional partial equivalence, i.e., equivalence under certain input constraints. Moreover, SymDiff also allows automated extraction of the constraints and reports them to the user. The goal of *differential symbolic execution* [14] is to show equivalence of the two versions using symbolic execution. If the versions are not equivalent, a behavioral delta is constructed as a feedback for the user. In [7], a technique called *regression verification* for deciding partial equivalence of programs using model checking is introduced. As well as eVolCheck, regression verification starts with a syntactic difference check identifying the modified functions. Then the call graph is traversed starting from the leaves. During the traversal, old and new versions of each visited and possibly affected function is checked for equivalence. In this check, any called functions are abstracted using the same uninterpreted functions.

If we compare these approaches to eVolCheck, the fundamental difference is that eVolCheck does not care about equivalence. It only checks that no errors sneak in the code with the upgrade. This means that the equivalence-based approaches report all the nonequivalent behavior to the user, flooding the user with information in the process. In contrast, eVolCheck reports only the bugs added to the code, which we believe the users are really interested in. Another related benefit is that eVolCheck may skip processing parts of the code base (which could be hefty) that do not affect correctness of the upgrade.

In the context of compositional directed testing (a.k.a. white-box fuzzing), some authors study effects of upgrades on function summaries [6] with the goal to identify the affected summaries unusable for analysis of the new version. With the unusable summaries removed, the preserved ones are employed in the actual analysis as a second step. When compared, eVolCheck uses over-approximative interpolation-based function summaries and performs the actual verification during the analysis not separately.

7 Conclusion

This paper presented the incremental upgrade checker, eVolCheck along with its integration into the Eclipse development environment. This is the first tool which uses interpolation-based function summaries to localize and speed up the upgrade check. The tool was evaluated on a range of industrial and academic examples, and in the most cases showed notable speedup with relation to verification from scratch. In future, we would like to explore the possibility to use the information regarding the failed and successful intermediate summary checks in order to guide the user in finding the root cause of the error, e.g., by emphasizing portions of the reported error trace corresponding to the failed intermediate summary checks. In addition, we consider integration with a versioning system (e.g., SVN), which would allow further integration into the software development process.

Acknowledgments. We thank the following people for their valuable contribution during the work on this paper: Murillo Miranda Cristina Maria for her implementation work on the `Eclipse` plug-in, Antti Hyvärinen for his comments on usability and correcting our English, Michael Tautschnig for his help with `CProver` and `goto-cc` adjustments, and Pincette validators for assistance with the industrial test cases.

References

1. 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)
2. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
3. Sharygina, N., Chaki, S., Clarke, E., Sinha, N.: Dynamic Component Substitutability Analysis. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 512–528. Springer, Heidelberg (2005)
4. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental Algorithms for Interprocedural Analysis of Safety Properties. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 449–461. Springer, Heidelberg (2005)
5. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. of Symbolic Logic, 269–285 (1957)
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. Godlin, B., Strichman, O.: Regression verification. In: DAC 2009, pp. 466–471 (2009)
8. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Principles of Prog. Languages (POPL 2010), pp. 471–482. ACM (2010)
9. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme Model Checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)
10. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 712–717. Springer, Heidelberg (2012)
11. 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)
12. 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)
13. 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)
14. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: FSE 2008, pp. 226–237 (2008)
15. Rollini, S.F., Sery, O., Sharygina, N.: Leveraging Interpolant Strength in Model Checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 193–209. Springer, Heidelberg (2012)

16. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-Based Function Summaries in Bounded Model Checking. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 160–175. Springer, Heidelberg (2012)
17. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 203–207. Springer, Heidelberg (2012)
18. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In: FMCAD 2012, pp. 114–121. ACM (2012)
19. Sinha, N., Singhania, N., Chandra, S., Sridharan, M.: Alternate and Learn: Finding Witnesses without Looking All over. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 599–615. Springer, Heidelberg (2012)

Intertwined Forward-Backward Reachability Analysis Using Interpolants

Yakir Vizel[1], Orna Grumberg[1], and Sharon Shoham[2]

[1] Computer Science Department, The Technion, Haifa, Israel
[2] School of Computer Science, Academic College of Tel Aviv-Yaffo

Abstract. In this work we develop a novel SAT-based verification approach which is based on interpolation. The novelty of our approach is in extracting interpolants in both forward and backward manner and exploiting them for an *intertwined approximated forward and backward reachability analysis*. Our approach is also mostly *local* and avoids unrolling of the checked model as much as possible. This results in an efficient and complete SAT-based verification algorithm.

We implemented our algorithm and compared it with both McMillan's interpolation-based algorithm and with IC3, on real-life industrial designs as well as on examples from the HWMCC'11 benchmark. In many cases, our algorithm outperformed both methods.

1 Introduction

In this work we develop a novel SAT-based verification approach based on interpolation. The novelty of our approach is in extracting interpolants in both forward and backward manner and exploiting them for an *intertwined approximated forward and backward reachability analysis*. Our approach is also mostly *local* and avoids unrolling of the checked model as much as possible. This results in an efficient and complete SAT-based algorithm.

SAT-based model checking is a highly successful approach for the verification of real-life designs from both hardware and software domains. In its early days SAT-based model checking was used mostly for bug hunting. The introduction of *interpolation* [7] enabled an efficient complete algorithm, referred to as Interpolation-based model checking (ITP) [11].

ITP uses interpolation to extract an over-approximation of a set of reachable states from a proof of unsatisfiability, generated by a SAT-solver. This fact enables to perform a SAT-based reachability analysis. The set of reachable states computed by the reachability analysis is used by ITP to check if a system M satisfies a safety property AGp.

In [1] an alternative SAT-based algorithm, called IC3, is introduced. Similarly to ITP, IC3 also computes over-approximations of sets of reachable states. However, ITP unrolls the model in order to obtain more precise approximations. In many cases, this is a bottleneck of the approach. IC3, on the other hand, improves the precision of the approximations by performing many local checks that do not require unrolling.

Both ITP and IC3 compute over-approximations of the sets of states obtained by a *forward reachability analysis*. The forward analysis starts from the initial states of M, and iteratively computes predecessors while making sure that no bad state violating p is reached. Verification based on reachability can also be performed in a dual manner using a *backward reachability analysis*. The backward analysis starts from the states satisfying $\neg p$ and iteratively computes ancestors while making sure that no initial state is reached.

Traditionally, BDD-based verification methods [6] use both forward and backward analyses [3,13], while SAT-based methods mainly implement the forward one. Recently, a few works considered backward analysis in the context of SAT as well (e.g. [2,8]). Most such works use forward and backward analyses independently of each other, or use a weak combination of the two, such as replacing the role of the initial states in the backward analysis by the reachable states computed by a forward analysis.

In this work we propose an interpolation-based verification method that applies mostly local checks and avoids unrolling of the model as much as possible. Our approach combines approximated forward and backward analyses in a tight and intertwined way, and uses each of them to enhance the precision of the other. Thus, the tight combination of the two analyses replaces unrolling in enhancing the precision of the computed over-approximated sets of states.

Our work uses the observation that a single SAT check entails information both about states reachable from the initial states (via post-image operations) and about states that reach the bad-states (via pre-image operations). We exemplify this observation by examining the propositional formula $\mathit{INIT}(V) \wedge \mathit{TR}(V, V') \wedge \neg p(V')$ where *INIT* and $\neg p$ describe the sets of initial states and bad states, respectively, and $\mathit{TR}(V, V')$ describes the transition relation. If this formula is satisfiable, then there exists a path of length one from the initial states to the bad states. If it is unsatisfiable, then all states reachable from the initial states in one transition are a subset of p. This fact is often used in forward reachability. We now note that the unsatisfiability of this formula can be used in backward reachability as well. This can be done by interpreting it as “all states that can reach the bad states in one transition are disjoint from the initial states”.

We exploit this dual observation by extracting two different interpolants from the unsatisfiabe formula $\mathit{INIT}(V) \wedge \mathit{TR}(V, V') \wedge \neg p(V')$. The *forward interpolant* (the one used in ITP) provides an over-approximation of the post-image of *INIT* which is disjoint from $\neg p$. The *backward interpolant*, computed for the same formula when it is read backward, from right to left, provides an over-approximation of the pre-image of $\neg p$ which is disjoint from *INIT*.

We use the above observation as a key element in traversing the state space in a dual fashion, both forward from the initial states and backwards from the bad states.

Our algorithm, *Dual Approximated Reachability* (DAR), computes a *Forward Reachability Sequence* $\bar{F} = \langle F_0, F_1, \ldots \rangle$, and a *Backward Reachability Sequence* $\bar{B} = \langle B_0, B_1, \ldots \rangle$. The set F_i represents an over-approximation of the set of states which are reachable from *INIT* in exactly i transitions. Further, F_i is disjoint from $\neg p$. Similarly, B_i represents an over-approximation of the set of states that can reach $\neg p$ in exactly i transitions, and it is also disjoint from *INIT*. Thus, the existence of either $\bar{F}$ or $\bar{B}$ of length n ensures that no counterexample of length n exists in M.

The goal of DAR is to gradually strengthen (make more precise) and extend $\bar{F}$ and $\bar{B}$, until a counterexample is found or until one of $\bar{F}$ or $\bar{B}$ reaches a *fixpoint*, that is, no new states are found when the sequence is further extended. To do this, DAR employs local strengthening phases, assisted by a global strengthening phase, when needed. Only the global strengthening involves unrolling. Thus, the number of unrolling applications is limited. In addition, the depth of the unrolling is also limited.

Initially, $\bar{F} = \langle F_0 \rangle$ and $\bar{B} = \langle B_0 \rangle$, where $F_0 = \mathit{INIT}$ and $B_0 = \neg p$. At iteration n, we define the sequence $\Pi = \langle\ \mathit{INIT},\ F_1 \wedge B_n,\ F_2 \wedge B_{n-1},\ \ldots,\ F_n \wedge B_1,\ \neg p\ \rangle$. Π represents an over-approximation of the set of all possible paths from *INIT* to $\neg p$ of length $n+1$ in M. That is, Π over-approximates the set of all counterexamples in M of length $n+1$. DAR attempts to show that Π represents no counterexample.

The *local strengthening phase* checks whether there are in fact transitions between every two consecutive sets in Π. It turns out that this can be done by applying local checks of the form $F_i(V) \wedge \mathit{TR}(V,V') \wedge B_{n-i}(V')$. If such a formula is unsatisfiable, then no transition exists from $F_i \wedge B_{n-i+1}$ to its successor along Π, thus no counterexample of length $n+1$ exists. This can also be understood by observing that the unsatisfiability of $F_i(V) \wedge \mathit{TR}(V,V') \wedge B_{n-i}(V')$ means that the states reachable from the initial states in i transitions cannot reach B_{n-i} in one transition. Since B_{n-i} includes all states reaching $\neg p$ in $n-i$ transitions, no counterexample of length $n+1$ exists.

In this case, the forward interpolant of $F_i(V) \wedge \mathit{TR}(V,V') \wedge B_{n-i}(V')$ is used to strengthen F_{i+1} while the backward interpolant strengthens B_{n-i+1}. Strengthening is now propagated along $\bar{F}$ and $\bar{B}$. This reflects the fact that the components of one sequence are strengthened based on the components of the other everywhere along the sequences, making the analyses closely intertwined. Next, $\bar{F}$ and $\bar{B}$ are extended by initializing F_{n+1} to be the forward interpolant of $F_n(V) \wedge \mathit{TR}(V,V') \wedge B_0(V')$ and B_{n+1} to be the backward interpolant of $F_0(V) \wedge \mathit{TR}(V,V') \wedge B_n(V')$.

The *global strengthening phase* is applied when $F_i(V) \wedge \mathit{TR}(V,V') \wedge B_{n-i}(V')$ is satisfiable for *all* i. This implies that a transition exists between every two consecutive sets in Π, making local reasoning insufficient. We therefore gradually unroll the model M and check whether the states in $F_i \wedge B_{n-i+1}$ are *unreachable* from *INIT* via i transitions of M. Once we find such an i, the unrolling can stop. We are certain that no counterexample of length $n+1$ exists. We strengthen $\bar{F}$ up to depth i using an interpolation-sequence [10], and return to the local strengthening phase for further strengthening and for extending $\bar{F}$ and $\bar{B}$ to length $n+1$.

We implemented our DAR algorithm and compared it to both ITP and IC3, on real-life industrial designs as well as examples from the HWMCC'11 benchmark. In many cases, our algorithm outperformed both methods. We noticed that the number of iterations where global strengthening was needed, as well as the depth of the unrolling in the global strengthening phase is often smaller relative to the length of $\bar{F}$ and $\bar{B}$. This reflects the fact that our use of unrolling is limited.

To summarize, the novelty of our approach is twofold. It suggests a SAT-based intertwined forward-backward reachability analysis. Further, the reachability analysis is interpolation-based. Yet, it is mostly local and avoids unrolling as much as possible.

1.1 Related Work

Several works use interpolation in the context of model checking. Interpolation-based model checking (ITP) was initially introduced in [11]. Similarly to ITP, DAR also uses interpolation to compute over-approximated sets of reachable states. However, ITP computes interpolants based on an unrolled formula and increases unrolling to make the over-approximation more precise. DAR, on the other hand, mostly avoids unrolling and uses backward and forward interpolants from local checks for strengthening. In addition, ITP restarts when it finds a spurious counterexample, increasing the depth of unrolling. In contrast, DAR keeps strengthening the computed over-approximations from previous iterations. In [2] improvements for ITP are suggested. They implement a backward-traversal using interpolants. Unlike our method, their backward traversal is an adaptation of ITP and is not tightly integrated with the forward traversal.

The work in [8] is also based on ITP in the sense of computing interpolants based on unrolling of the model, where the depth of unrolling increases in each iteration. Their work integrates the use of forward and backward analyses: in each iteration the result of the backward analysis is used to restrict the initial states and the result of the forward analysis is used to restrict the bad states. Our approach, on the other hand, uses the result of the forward analysis to strengthen *all intermediate sets* of $\bar{B}$. Similarly the result of the backward analysis stregthens $\bar{F}$.

Interpolation-sequence, which extends the notion of an interpolant for a sequence of formulas has been proposed and used for model checking [10,12,14,4]. DAR makes a similar use of interpolation-sequence in its global strengthening phase. In contrast to the other methods, interpolation-sequence is not a key element of DAR since it is only applied occasionally. Further, it is applied to a restricted depth of unrolling.

The introduction of IC3 [1] suggested a different way to compute information about reachable states. Unlike interpolation-based approaches IC3 requires no unrolling and is based on inductive reasoning. The main difference between DAR and IC3 is in the way they strengthen the over-approximated sets of states. IC3 finds a state that can reach $\neg p$ and if it concludes that this state is not reachable, it tries to generalize this fact and removes more than just one state. DAR on the other hand finds an over-approximation of *all* states that can reach $\neg p$, rather than a single state. It then tries to prove that the entire set is unreachable. Also, when DAR fails to strengthen using local reasoning, it applies a limited unrolling in the global phase. On the other hand, IC3 can fall into state enumeration if generalization is not successful.

2 Preliminaries

Let V be a set of boolean variables. For $v \in V$, v' is used to denote the value of v after one time unit. The set of these variables is denoted by V'. In the general case V^i is used to denote the variables in V after i time units (thus, $V^0 = V$). For a formula η over V^i, we denote by $\eta[V^i \leftarrow V^j]$ the formula obtained from η when for each $v \in V$, v^i is replaced with v^j. We use $\eta(V^i)$ or simply $\eta^{\langle i \rangle}$ to denote $\eta[V \leftarrow V^i]$. In particular, $\eta' = \eta[V \leftarrow V']$. We will use $\mathcal{L}(\eta)$ to denote the variables appearing in η. From now on, all formulas we refer to are *propositional formulas*, unless stated otherwise.

Definition 1. *A* finite transition system *is a triple* $M = (V, \text{INIT}, \text{TR})$ *where* V *is a set of boolean variables,* $\text{INIT}(V)$ *is a formula over* V*, describing the initial states, and* $\text{TR}(V, V')$ *is a formula over* V *and the next-state variables* V'*, describing the transition relation.*

An assignment s assigning values from $\{0, 1\}$ to V defines a *state* in M. A formula over V represents a set of states which consists of all the satisfying assignments of the formula. We refer to a formula η over V as a set of states and therefore use the notation $s \in \eta$ for states represented by η. Similarly, a formula η over V, V' represents a set of pairs of states, and we write $(s, s') \in \eta$ for pairs in the set.

A *path* of length n in M is a sequence of states $\pi = s_0, \ldots, s_n$ s.t. $s_0 \in \mathit{INIT}$ and for all $0 \leq i < n$, $(s_i, s_{i+1}) \in \mathit{TR}$. Let AGp be a safety property, where p is a formula over V. A path $\pi = s_0, \ldots, s_n$ in M is a *counterexample* of length n for AGp if $s_n \models \neg p$.

Let Q be a formula over V. The *post-image* of Q w.r.t. M is the set of all states reachable from Q in one transition, defined by the formula $\exists V[Q(V) \wedge \mathit{TR}(V, V')]$ (note that this formula is defined over V'). The *pre-image* of Q w.r.t. M is the set of all states that can reach Q in one transition, defined by $\exists V'[\mathit{TR}(V, V') \wedge Q(V')]$.

Definition 2. *Let* M *be a transition system and* φ *and* ψ *formulas over* V*. The formula* $\Gamma_M(\varphi, \psi) = \varphi(V) \wedge \text{TR}(V, V') \wedge \psi(V')$ *is a* local reachability check *w.r.t.* M*,* φ*,* ψ*.*

Whenever M is clear from the context we omit M and write $\Gamma(\varphi, \psi)$.

Let (ϕ^-, ϕ^+) be a pair of formulas. If $\phi^- \wedge \phi^+$ is unsatisfiable, then by [7] we know that there exists an interpolant, defined as follows.

Definition 3 (Interpolant). *Let* $\phi^- \wedge \phi^+ \equiv \bot$ *be an unsatisfiable formula. An* interpolant *for* $\phi^- \wedge \phi^+$*, denoted* $I(\phi^-, \phi^+)$*, is a formula* I *s.t. (i)* $\phi^- \Rightarrow I$*, (ii)* $I \wedge \phi^+ \equiv \bot$*, and (iii)* $\mathcal{L}(I) \subseteq \mathcal{L}(\phi^-) \cap \mathcal{L}(\phi^+)$*.*

A similar property holds for conjunctions of more than 2 formulas [10,14]:

Definition 4 (Interpolation-Sequence). *Let* $\langle A_1, \ldots, A_n\rangle$ *be a sequence of formulas s.t.* $\bigwedge_{i=1}^{n} A_i \equiv \bot$*. An* interpolation-sequence *for* $\langle A_1, \ldots, A_n\rangle$ *is a sequence* $\langle I_0, I_1, \ldots, I_n\rangle$ *of formulas s.t.: (i)* $I_0 \equiv \top$ *and* $I_n \equiv \bot$*, (ii) For every* $0 \leq j < n$*,* $I_j \wedge A_{j+1} \Rightarrow I_{j+1}$*, and (iii) For every* $0 < j < n$*,* $\mathcal{L}(I_j) \subseteq \mathcal{L}(A_1, \ldots, A_j) \cap \mathcal{L}(A_{j+1}, \ldots, A_n)$*.*

3 Using Interpolants for Forward and Backward Analysis

3.1 Forward and Backward Interpolants

Interpolation is typically used in model checking in order to compute over-approximated sets of reachable states [11,12,14].

Let R and Q be propositional formulas over V representing sets of states, and let $\mathit{TR}(V, V')$ be a transition relation. Suppose we would like to know if the post image of R is disjoint from Q. This property can be checked by checking the formula $\Gamma(R, Q) = R(V) \wedge \mathit{TR}(V, V') \wedge Q(V')$ for unsatisfiability. If the formula is unsatisfiable then the answer is yes, meaning that Q is not reachable from R in one step. Moreover, consider $\phi^- = R(V) \wedge \mathit{TR}(V, V')$ and $\phi^+ = Q(V')$. An interpolant $I = I(\phi^-, \phi^+)$ satisfies $R(V) \wedge \mathit{TR}(V, V') \Rightarrow I(V')$ and $I(V') \wedge Q(V') \equiv \bot$. Therefore, I represents an over approximation of the post-image of R, and it is also disjoint from Q.

The unsatisfiability of the formula $\Gamma(R, Q) = R(V) \wedge \mathit{TR}(V, V') \wedge Q(V')$ can also be interpreted in a different manner, shedding light on the pre-image of Q. More precisely, the unsatisfiability of the formula states that the pre-image of Q is disjoint from R. This view leads to a different way of using interpolation in this setting. For the backward interpretation, we now define $\phi^- = \mathit{TR}(V, V') \wedge Q(V')$ and $\phi^+ = R(V)$. Again, since $\phi^- \wedge \phi^+$ is unsatisfiable, an interpolant I exists. Formally $\mathit{TR}(V, V') \wedge Q(V') \Rightarrow I(V)$, therefore I is an over-approximation of the pre-image of Q. Moreover, $I \wedge R$ is unsatisfiable and therefore I is disjoint from R.

We conclude that interpolation gives us a way to approximate both post-image and pre-image computations. Formally, we define forward and backward interpolants:

Definition 5 (Forward and Backward Interpolants). *Let R and Q be propositional formulas over V s.t. $\Gamma(R, Q) \equiv \bot$. The* forward interpolant *of $\Gamma(R, Q)$, denoted $\mathit{FI}(R, Q)$, is $I(R(V) \wedge \mathrm{TR}(V, V'), Q(V'))[V' \leftarrow V]$. The* backward interpolant *of $\Gamma(R, Q)$, denoted $\mathit{BI}(R, Q)$, is $I(\mathrm{TR}(V, V') \wedge Q(V'), R(V))$.*

Note that $I(R(V) \wedge \mathit{TR}(V, V'), Q(V'))$ is defined over V'. Therefore, we substitute V' for V in the definition of a forward interpolant. As explained above:

Lemma 1. *$\mathit{FI}(R, Q)$ over-approximates the post-image of R, and is disjoint from Q. Similarly, $\mathit{BI}(R, Q)$ over-approximates the pre-image of Q, and is disjoint from R.*

3.2 Forward and Backward Reachability Sequences

Our model checking algorithm for safety properties, described in Sec. 4, uses forward and backward interpolants for the computation of over-approximated sets of forward and backward reachable states. Technically, we consider both forward and backward reachability approximations:

Definition 6. *A* Forward Reachability Sequence (FRS) *of length n w.r.t. M and a property AGp is a sequence $\bar{F}_{[n]} = \langle F_0, F_1, \ldots, F_n \rangle$ of sets of states s.t.*

- $F_0 = \mathrm{INIT}$
- $F_i(V) \wedge \mathrm{TR}(V, V') \Rightarrow F_{i+1}(V')$ *for* $0 \leq i < n$
- $F_i \Rightarrow p$ *for* $0 \leq i \leq n$.

Definition 7. *A* Backward Reachability Sequence (BRS) *of length n w.r.t. M and a property AGp is a sequence $\bar{B}_{[n]} = \langle B_0, B_1, \ldots, B_n \rangle$ of sets of states s.t.*

- $B_0 = \neg p$.
- $B_{i+1}(V) \Leftarrow \mathrm{TR}(V, V') \wedge B_i(V')$ *for* $0 < i \leq n$.
- $B_i \Rightarrow \neg\mathrm{INIT}$ *for* $0 \leq i \leq n$.

```
function DAR($M$,$p$)
    if $INIT \wedge \neg p$ == SAT then
        return $cex$
    end if
    $\bar{F} = \langle F_0 = INIT \rangle$, $\bar{B} = \langle B_0 = \neg p \rangle$
    $n = 0$
    while $!\bar{F}$.FIXPOINT()$\wedge!\bar{B}$.FIXPOINT() do
        if LOCSTRENGTHEN($\bar{F}, \bar{B}, n$) == $false$ then
            if GLBSTRENGTHEN($\bar{F}, \bar{B}, n$) == $false$ then
                return $cex$
            end if
        end if
        $n = n + 1$
    end while
    return Verified
end function
```

Fig. 1. Dual Approximated Reachability

When n is clear from the context, we simply use $\bar{F}$ and $\bar{B}$. The second condition in Def. 6 (Def. 7) states that F_{i+1} (B_{i+1}) is an over-approximation of the post(pre)-image of F_i (B_i) w.r.t. M. We conclude that F_i over-approximates the set of states reachable from *INIT* in i steps, and B_i over-approximates the set of states reaching a violation of p in i steps. The following properties hold for FRS and BRS:

Lemma 2. *A FRS (BRS) of length n exists iff there is no counterexample of length $\leq n$.*

Definition 8 (Fixpoint). *A FRS $\bar{F}_{[n]}$ is at* fixpoint *if there is $0 < k \leq n$ s.t. $F_k \Rightarrow \bigvee_{i=0}^{k-1} F_i$. Similarly, a BRS $\bar{B}_{[n]}$ is at* fixpoint *if there is $0 < k \leq n$ s.t. $B_k \Rightarrow \bigvee_{i=0}^{k-1} B_i$.*

Lemma 3. *Given a FRS $\bar{F}$ and a BRS $\bar{B}$, if $\bar{F}$ or $\bar{B}$ is at fixpoint then $M \models AGp$.*

Note that a fixpoint in one of the sequences suffices to conclude that $M \models AGp$.

4 Dual Approximated Reachability

In this section we describe our Dual Approximated Reachability (DAR) algorithm for model checking safety properties. DAR computes over-approximated sets of reachable states for both forward and backward reachability analysis by means of a FRS and a BRS, using interpolants. The computations are intertwined where each of them is used to make the other tighter. DAR avoids unrolling of the transition system unless it is really needed.

Technically, DAR computes a FRS $\bar{F}$ and a BRS $\bar{B}$ and gradually extends them until either a counterexample is found or a fixpoint is reached on either $\bar{F}$ or $\bar{B}$. Since the state-space of M is finite, one of the above is bound to happen, which ensures that:

Theorem 1. *Given a model M and a safety property $\varphi = AGp$, DAR always terminates. Moreover, $M \models \varphi$ if and only if DAR returns "Verified".*

We now describe DAR in detail. The pseudocode of DAR appears in Fig. 1.

Initialization of DAR (lines 2-5) starts by checking the formula $\mathit{INIT} \wedge \neg p$. If this formula is unsatisfiable, the initial states of M satisfy the property. If not, a counterexample exists. In the former case, DAR initializes $\bar{F} = \langle F_0 = \mathit{INIT} \rangle$ and $\bar{B} = \langle B_0 = \neg p \rangle$. Clearly $\bar{F}$ and $\bar{B}$ are FRS and BRS, respectively.

The iterative part of DAR (lines 8-13) then gradually extends and strengthens $\bar{F}$ and $\bar{B}$ s.t. they remain a FRS and a BRS respectively. As ensured by Lemma 2, this is possible as long as no counterexample of the corresponding length exists. In the following, we describe a single iteration of DAR, strengthening and extending $\bar{F}$ and $\bar{B}$, or reporting a counterexample.

4.1 First Iteration of DAR

Let us first present the first iteration of DAR. Recall that initially $\bar{F} = \langle F_0 = \mathit{INIT} \rangle$ and $\bar{B} = \langle B_0 = \neg p \rangle$. DAR then checks the formula $F_0 \wedge \mathit{TR} \wedge B_0' = \mathit{INIT} \wedge \mathit{TR} \wedge \neg p'$ for satisfiability. In case this formula is satisfiable a counterexample of length one exists. Otherwise, the unsatisfiability of $\mathit{INIT} \wedge \mathit{TR} \wedge \neg p'$ entails information both about the post-image of *INIT* and about the pre-image of $\neg p$. Accordingly, we extend $\bar{F}$ with $F_1 = \mathrm{FI}(F_0, B_0)$ and $\bar{B}$ with $B_1 = \mathrm{BI}(F_0, B_0)$. Due to the properties of the interpolants, the sequences $\bar{F} = \langle F_0, F_1 \rangle$ and $\bar{B} = \langle B_0, B_1 \rangle$ are FRS and BRS respectively.

4.2 General Iteration of DAR

Let us now discuss a general iteration $n+1$. Consider the FRS $\bar{F}_{[n]} = \langle F_0, F_1, \ldots, F_n \rangle$ and the BRS $\bar{B}_{[n]} = \langle B_0, B_1, \ldots B_n \rangle$ obtained at iteration n. The goal of iteration $n+1$ is to check if a counterexample of length $n+1$ exists, and if not, extend these sequences to length $n+1$ s.t. they remain a FRS and a BRS.

The combination of $\bar{F}_{[n]}$ and $\bar{B}_{[n]}$ provides an approximate description of all possible counterexamples of length $n+1$ in M. Namely, recall that F_i over-approximates the set of all states reachable from *INIT* in i steps. Similarly, B_j over-approximates the set of all states that can reach $\neg p$ in j steps. Their intersection, $F_i \wedge B_j$ therefore over-approximates the set of all states that are both reachable from *INIT* in i steps and can reach $\neg p$ in j steps. These are states that appear in the i-th step of a counterexample of length $i + j$. In particular, when we align $\bar{F}$ and $\bar{B}$ one against the other, conjoining F_i with B_{n-i+1}, we obtain an over-approximation of the set of all states that appear in the i-th step of a counterexample of length $n+1$. The sequence

$$\Pi(\bar{F}_{[n]}, \bar{B}_{[n]}) = \langle \mathit{INIT}, F_1 \wedge B_n, F_2 \wedge B_{n-1}, \ldots, F_n \wedge B_1, \neg p \rangle$$

therefore over-approximates the set of *all* counterexamples of length $n+1$.

We refer to the sequence $\Pi(\bar{F}_{[n]}, \bar{B}_{[n]})$ as an *approximated Counterexample* (aCEX). Whenever clear from the context we write Π and refer to the i-th element in the sequence as Π_i. A sequence of states $s_0, \ldots, s_{n+1}$ in M *matches* Π if for every $0 \leq i \leq n+1$, $s_i \in \Pi_i$. Formally, Π has the following property.

Lemma 4. *Let $\pi = s_0, \ldots, s_{n+1}$ be a counterexample in M. Then, π matches Π.*

By Lemma 4, checking if a counterexample exists amounts to checking if some path matches Π. Such a path is necessarily a counterexample of length $n+1$. If such a path exists, we say that Π is *valid*.

DAR first attempts to check for (in)validity of the aCEX using local checks in a *local strengthening phase*. If this fails, DAR moves on to the *global strengthening phase* that applies global checks. In both phases, if the invalidity of the aCEX is established, the FRS and BRS are strengthened and extended into a FRS and a BRS of length $n+1$. Otherwise, the aCEX is found to be valid and a counterexample of length $n+1$ is obtained in the process.

Local Strengthening Phase. The local strengthening phase aims at checking if Π is *locally invalid*, which provides a sufficient condition for its invalidity.

Definition 9. *Π is* locally invalid *if there exists* $0 \le i \le n$ *s.t.* $\Gamma(\Pi_i, \Pi_{i+1}) \equiv \bot$.

Lemma 5. *If Π is locally invalid, then it is also invalid.*

In order to check if Π is locally invalid, we use the following observation.

Lemma 6. *Let $\bar{F}_{[n]}$ be a FRS, $\bar{B}_{[n]}$ be a BRS, and $1 \le i \le n$. Then $\Gamma(F_i \wedge B_{n-i+1}, F_{i+1} \wedge B_{n-i}) \equiv \Gamma(F_i, B_{n-i})$. Similarly, $\Gamma(\text{INIT}, F_1 \wedge B_n) \equiv \Gamma(F_0, B_n)$, and $\Gamma(F_n \wedge B_1, \neg p) \equiv \Gamma(F_n, B_0)$. We conclude that for every $0 \le i \le n$, $\Gamma(\Pi_i, \Pi_{i+1}) \equiv \bot$ iff $\Gamma(F_i, B_{n-i}) \equiv \bot$.*

Lemma 6 follows from the property of a FRS, where $F_i \wedge TR \Rightarrow F'_{i+1}$, and the property of a BRS, where $B_{n-i+1} \Leftarrow TR \wedge B'_{n-i}$. Lemma 6 implies that if there exists $0 \le i \le n$ s.t. $\Gamma(F_i, B_{n-i}) \equiv \bot$, then the aCEX is locally invalid and hence invalid. This can also be understood intuitively, as the above means that the (over-approximated) set of states reachable from *INIT* in i steps and the (over-approximated) set of states that can reach $\neg p$ in $n-i$ steps are not reachable from one another in one step. This means that altogether $\neg p$ is not reachable from *INIT* in $i + (n-i) + 1 = n+1$ steps, and hence no counterexample of length $n+1$ exists.

In the local strengthening phase, DAR therefore searches for an index $0 \le i \le n$ s.t. $\Gamma(F_i, B_{n-i}) \equiv \bot$. It starts by checking the formula $\Gamma(F_n, B_0)$, setting $i = n$. In case it is satisfiable, DAR starts to iteratively go backwards along $\bar{F}$ and $\bar{B}$ decreasing i by 1. The traversal continues until either $\Gamma(F_i, B_{n-i})$ turns out to be unsatisfiable for some $0 \le i \le n$ or until $\Gamma(F_0, B_n)$ is found to be satisfiable.

If an index i is found s.t. $\Gamma(F_i, B_{n-i}) \equiv \bot$, then the aCEX is locally invalid and by Lemma 5 we conclude that no counterexample of length $n+1$ exists. Moreover, in this case, the FRS and BRS are locally and gradually strengthened and extended as follows.

Iterative Local Strengthening: Iterative local strengthening is reached when it is already known that no counterexample of length $n+1$ exists. Thus, as Lemma 2 ensures, there exist a FRS and BRS of length $n+1$. However, $\bar{F}_{[n]}$ and $\bar{B}_{[n]}$ cannot necessarily be extended immediately. For example, if $\Gamma(F_n, B_0) = F_n(V) \wedge TR(V, V') \wedge \neg p(V') \not\equiv \bot$, then no F_{n+1} can be obtained s.t. $F_n(V) \wedge TR(V, V') \Rightarrow F_{n+1}(V')$ and in addition $F_{n+1} \Rightarrow p$. On the other hand, if $\Gamma(F_n, B_0) \equiv \bot$ then F_{n+1} can be initialized using

FI(F_n, B_0) while maintaining the properties of a FRS (similarly to the initialization of F_1). Dually, if $\Gamma(F_0, B_n) \not\equiv \bot$, then no extension of $\bar{B}_{[n]}$ is possible, while if $\Gamma(F_0, B_n) \equiv \bot$, we can set $B_{n+1} = \text{BI}(F_0, B_n)$. We therefore first strengthen the components of $\bar{F}_{[n]}$ and $\bar{B}_{[n]}$ until $\Gamma(F_n, B_0) \equiv \bot$ and $\Gamma(F_0, B_n) \equiv \bot$, which is a necessary and sufficient condition for extending $\bar{F}$ and $\bar{B}$.

Recall that $\Gamma(F_i, B_{n-i}) \equiv \bot$ for some $0 \leq i \leq n$. This means that even though the components of $\bar{F}_{[n]}$ and $\bar{B}_{[n]}$ may not be precise enough to enable their extension, they are precise enough at least in one place that allowed us to conclude that no counterexample of length $n + 1$ exists. DAR uses this "local" precision to strengthen the entire sequences, as described below.

In order to simplify the references to the indices, we replace the use of i and $n - i$ by $0 \leq i, j \leq n$ s.t. $i + j = n$. Therefore $\Gamma(F_i, B_j) \equiv \bot$ for some $0 \leq i, j \leq n$ s.t. $i + j = n$. This ensures that there exists a forward interpolant FI(F_i, B_j), as well as a backward interpolant BI(F_i, B_j). We can therefore perform a *local strengthening step* updating F_{i+1} and B_{j+1}:

Definition 10. *Let $\bar{F}_{[n]}$ be a FRS and $\bar{B}_{[n]}$ be a BRS s.t. $\Gamma(F_i, B_j) \equiv \bot$ for some $0 \leq i, j \leq n$ s.t. $i + j = n$. A* forward strengthening step at (i, j) *strengthens $\bar{F}_{[n]}$: If $i < n$, $F_{i+1} = F_{i+1} \wedge$ FI(F_i, B_j). A* backward strengthening step at (i, j) *strengthens $\bar{B}_{[n]}$: If $j < n$, $B_{j+1} = B_{j+1} \wedge$ BI(F_i, B_j).*

We refer to $i, j < n$ since F_{n+1} and B_{n+1} are not yet defined and therefore cannot be updated. The strengthening propagates the unsatisfiability of $\Gamma(F_i, B_j)$ one step forward and one step backward while maintaining the properties of a FRS and a BRS:

Lemma 7. *Let $\bar{F}_{[n]}$ and $\bar{B}_{[n]}$ be the result of a forward or backward strengthening step at (i, j) s.t. $i + j = n$. Then $\bar{F}_{[n]}$ and $\bar{B}_{[n]}$ remain FRS and BRS resp. In addition:*

- *For a forward strengthening step, if $i < n$, $\Gamma(F_{i+1}, B_{j-1}) \equiv \bot$.*
- *For a backward strengthening step, if $j < n$, $\Gamma(F_{i-1}, B_{j+1}) \equiv \bot$.*

Lemma 7 implies that if $\Gamma(F_i, B_j) \equiv \bot$ for *some* $0 \leq i, j \leq n$ s.t. $i + j = n$, then by iterating the forward and backward strengthening steps, we can eventually ensure that $\Gamma(F_i, B_j) \equiv \bot$ for *every* $0 \leq i, j \leq n$ s.t. $i + j = n$, and in particular for $i = 0, j = n$ and $i = n, j = 0$. Thus, we apply an *iterative local strengthening* starting from (i, j):

Definition 11 (Iterative Local Strengthening). *Let $0 \leq i, j \leq n$ be indices s.t. $i + j = n$ and $\Gamma(F_i, B_j) \equiv \bot$.* Iterative local strengthening from (i, j) *performs:*

1. *Forward strengthening steps starting at (i, j), proceeding forward while increasing i and decreasing j until $(n - 1, 1)$ (strengthening $F_{i+1}, \ldots, F_n$), and*
2. *Backward strengthening steps starting at (i, j), proceeding backward while increasing j and decreasing i until $(1, n - 1)$ (strengthening $B_{j+1}, \ldots, B_n$), and*
3. *Finally, once $\Gamma(F_n, B_0) \equiv \bot$, F_{n+1} is initialized by FI(F_n, B_0). Similarly, once $\Gamma(F_0, B_n) \equiv \bot$, B_{n+1} is initialized by BI(F_0, B_n).*

Lemma 8. *Let $0 \leq i, j \leq n$ be indices s.t. $i + j = n$ and $\Gamma(F_i, B_j) \equiv \bot$. Iterative local strengthening from (i, j) terminates with a FRS and a BRS of length $n + 1$.*

Iterative local strengthening uses the BRS for the strengthening of the FRS and vice versa, demonstrating how each of them is used to make the other over-approximation tighter. The complete local strengthening procedure is described in Fig. 2.

```
17: function LocStrengthen($\bar{F}, \bar{B}, n$)
18:     $i$ = FindStrengthen($\bar{F}, \bar{B}, n$)
19:     if $i == -1$ then
20:         // No local strengthening
21:         // point was found
22:         //Move to GlbStrengthen
23:         return $false$
24:     else
25:         IterLS($\bar{F}, \bar{B}, n, i, n - i$)
26:         return $true$
27:     end if
28: end function
```

(a) Local Strengthening

```
29: function IterLS($\bar{F}, \bar{B}, n, i, j$)
30:     while $i < n$ do
31:         $F_{i+1} = F_{i+1} \wedge \mathrm{FI}(F_i, B_{n-i})$
32:         $i = i + 1$
33:     end while
34:     $\bar{F}$.Add(FI($F_n, B_0$))
35:     while $j < n$ do
36:         $B_{j+1} = B_{j+1} \wedge \mathrm{BI}(F_{n-j}, B_j)$
37:         $j = j + 1$
38:     end while
39:     $\bar{B}$.Add(BI($F_0, B_n$))
40: end function
```

(b) Iterative Local Strengthening

Fig. 2. Local strengthening procedures

Global Strengthening Phase. We now consider the case where $\Gamma(F_i, B_{n-i}) \not\equiv \bot$ for every $0 \leq i \leq n$ in $\bar{F}_{[n]}$ and $\bar{B}_{[n]}$. By Lemma 6, this means that there is a real transition between every pair of consecutive sets in the aCEX Π, making local strengthening inapplicable since the aCEX is not locally invalid. Clearly this does not imply that the aCEX is valid, and further checks are needed. We therefore turn to examine the (in)validity of the aCEX in a more global manner.

Similarly to the principle used in CEGAR [5] for an *abstract* counterexample, here too, if the aCEX Π is invalid, there exists a minimal index $i \leq n + 1$ representing the minimal prefix of the aCEX that has no matching path in M. We therefore wish to search for such an index, if it exists. The search starts from the prefix Π_0, Π_1, Π_2 (since $\langle \Pi_0, \Pi_1 \rangle$ is necessarily valid) and extends it gradually. In the i-th step (starting from $i = 2$), the goal is to check if $\Pi_0 \wedge \mathit{TR} \wedge \Pi_1' \wedge \mathit{TR} \wedge \Pi_2'' \ldots \wedge \mathit{TR} \wedge \Pi_i^{\langle i \rangle}$ (*) is satisfiable, meaning that a matching path to the prefix $\Pi_0, \ldots, \Pi_i$ exists in M.

Recall that for $i \leq n$, (*) is actually the formula $\mathit{INIT} \wedge \mathit{TR} \wedge (F_1 \wedge B_n)' \wedge \mathit{TR} \wedge (F_2 \wedge B_{n-1})'' \wedge \ldots \wedge \mathit{TR} \wedge (F_i \wedge B_{n-i+1})^{\langle i \rangle}$. For $i = n + 1$ the last conjunct consists of B_0 only (without an $\bar{F}$-component). In fact, since in a FRS $F_j \wedge \mathit{TR} \Rightarrow F_{j+1}'$, then removing all $\bar{F}$ components except for the first (*INIT*) results in an equivalent formula. Similarly, since in a BRS $B_{j+1} \Leftarrow \mathit{TR} \wedge B_j'$, removing all $\bar{B}$ components but the last (B_{n-i+1}) again results in an equivalent formula. This simplifies the formula as follows.

Lemma 9. *For every* $2 \leq i \leq n + 1$*:* $\Pi_0 \wedge \mathrm{TR} \wedge \Pi_1' \wedge \mathrm{TR} \wedge \Pi_2'' \wedge \ldots \wedge \mathrm{TR} \wedge \Pi_i^{\langle i \rangle}$ *is equivalent to* $\mathrm{INIT} \wedge \mathrm{TR} \wedge \mathrm{TR} \wedge \ldots \wedge \mathrm{TR} \wedge B_{n-i+1}^{\langle i \rangle}$.

DAR therefore checks formulas of the form $\mathit{INIT} \wedge \mathit{TR} \wedge \ldots \wedge \mathit{TR} \wedge B_{n-i+1}^{\langle i \rangle}$ starting from $i = 2$. It keeps on adding transitions until either the formula becomes unsatisfiable, or until $i = n + 1$ is reached (ending with $B_0 = \neg p$). If the formula is still satisfiable for $i = n + 1$, a counterexample is found and DAR terminates.

If for some $2 \leq i \leq n+1$, $\mathit{INIT} \wedge \mathit{TR} \wedge \ldots \wedge \mathit{TR} \wedge B_{n-i+1}^{\langle i \rangle}$ turns out to be unsatisfiable, making the aCEX invalid, then first $\bar{F}_{[n]}$ is strengthened:

```
41: function GlbStrengthen($\bar{F}, \bar{B}, n$)
42:     for $i = 2 \to n + 1$ do        // $n = 0$ does not go into the loop
43:         if $INIT \wedge TR \ldots \wedge TR \wedge B^{\langle i \rangle}_{n-i+1}$ == UNSAT then
44:             $\bar{I}$ = GetInterpolationSeq()
45:             for $j = 1 \to \min\{i, n\}$ do
46:                 $F_j = F_j \wedge I_j$
47:             end for
48:             IterLS($\bar{F}, \bar{B}, n, i - 1, n - i + 1$)
49:             return $true$
50:         end if
51:     end for
52:     return $false$        // counterexample
53: end function
```

Fig. 3. Global strengthening procedure

Definition 12. *Let* INIT $\wedge$ TR $\wedge \ldots \wedge$ TR $\wedge B^{\langle i \rangle}_{n-i+1} \equiv \bot$ *for some* $2 \leq i \leq n + 1$, *and let* $\langle I_0, I_1, \ldots, I_{i+1} \rangle$ *be an interpolation-sequence for* $\langle A_1 =$ INIT $\wedge$ TR, $A_2 =$ TR, $\ldots, A_i =$ TR, $A_{i+1} = B^{\langle i \rangle}_{n-i+1} \rangle$. *A* global strengthening step at index i *strengthens* F_j *for every* $1 \leq j \leq \min\{i, n\}$ *by setting* $F_j = F_j \wedge I_j$.

The condition $1 \leq j \leq \min\{i, n\}$ ensures that if $i = n+1$, strengthening is applied only up to F_n since F_{n+1} is not yet defined[1]. The following Lemma, along with Lemma 6 ensures that after a global strengthening step, the strengthened aCEX is locally invalid.

Lemma 10. *Let* $\bar{F}_{[n]}$ *be the result of a global strengthening step at index* $2 \leq i \leq n+1$. *Then* $\bar{F}_{[n]}$ *remains a FRS. In addition,* $\Gamma(F_{i-1}, B_{n-i+1}) \equiv \bot$.

DAR now uses iterative local strengthening from $(i-1, n-i+1)$ (Def. 11) to strengthen $F_i, \ldots, F_n$ and $B_{n-i+2}, \ldots, B_n$[2], as well as initialize F_{n+1} and B_{n+1}. The complete global strengthening procedure is described in Fig. 3.

5 Experimental Results

To implement DAR we collaborated with *Jasper Design Automation*[3]. We measured the efficiency of DAR by comparing it against two top-tier methods: ITP and IC3. We used Jasper's formal verification platform in order to implement DAR, ITP and IC3.

[1] If a global strengthening step is performed at $i = n + 1$, then F_{n+1} can be initialized to I_{n+1}.

[2] Note that instead of performing a local strengthening of $\bar{B}$ as part of the iterative local strengthening, an interpolation-sequence $\langle J_0, J_1, \ldots, J_{i+1} \rangle$ for $\langle A_1 = TR \wedge B^{\langle i+1 \rangle}_{n-i}, A_2 = TR, \ldots A_i = TR, A_{i+1} = INIT \rangle$ can be used to strengthen $B_{n-i+1}, \ldots, B_n$ by setting $B_{n-i+j} = B_{n-i+j} \wedge J_j$ for $1 \leq j \leq i$, and to initialize B_{n+1} to J_{i+1}. In this case, iterative local strengthening will be performed only forward, updating $\bar{F}$ only. For simplicity of the presentation, we use iterative local strengthening both forward and backward instead of using an interpolation-sequence for the backward update.

[3] An EDA company: `http://www.jasper-da.com`

Table 1. Parameters of the experiments. *Name*: name of the property; ♯*Vars*: number of state variables in the cone of influence; *Status*: *true* - verified property, *false* - indicates a counterexample; *D*: *convergence depth* representing the number of over-approximated sets of states computed when the algorithm converges (for ITP, the number of sets computed for the last bound used, and for DAR, the length of $\bar{F}$ and $\bar{B}$); *MaxU*: *maximum unrolling* used during verification; ♯*GS*: number of times Global Strengthening is used in DAR; GS$_R$: ratio between iterations using global strengthening to the total number of iterations; *Time[s]*: *time* in seconds. Minimal runtime appears in boldface. Properties above the full line are from real industrial designs. The rest are from HWMCC'11.

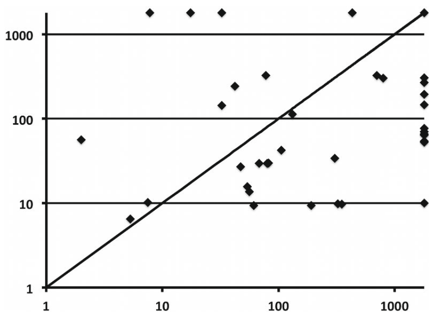
			IC3		ITP			DAR				
Name	♯Vars	Status	D	Time[s]	D	MaxU	Time[s]	D	MaxU	♯GS	GS$_R$	Time[s]
Ind_1	11854	true	46	799	41	28	1138	49	35	21	0.42	**303**
Ind_2	11854	true	44	701	41	28	1148	49	35	18	*0.36*	**326**
Ind_3	11866	true	11	82	5	2	**19.1**	11	8	4	*0.33*	29.9
Ind_4	11877	true	NA	TO	33	12	307	36	30	18	0.48	**194**
Ind_5	11871	false	NA	TO	NA	20	88	19	20	10	0.5	**77**
Ind_6	11843	false	NA	TO	NA	19	77	18	19	9	0.47	**70**
Ind_7	1247	true	6	**1.5**	3	2	2	17	5	9	0.5	56.3
Ind_8	1247	true	7	**7.8**	17	23	1250	NA	NA	NA	NA	TO
Ind_9	449	true	337	**78**	NA	NA	TO	45	12	22	0.48	327
Ind_{10}	331	true	458	305	NA	NA	TO	26	11	15	0.56	**33.9**
Ind_{11}	330	true	419	132	NA	NA	TO	38	12	19	0.49	**113**
Ind_{12}	450	true	22	**32.5**	NA	NA	TO	NA	NA	NA	NA	TO
Ind_{13}	3837	false	NA	TO	NA	68	369	67	68	33	0.48	**305**
Ind_{14}	3837	false	NA	TO	NA	69	487	68	69	25	*0.36*	**269**
Ind_{15}	3836	true	6	42	4	2	**2.3**	70	64	32	0.45	243
Ind_{16}	11860	true	9	32.5	5	2	**11.4**	33	32	16	0.47	144
Ind_{17}	11878	true	14	68	7	4	**18.4**	11	8	4	*0.33*	29.5
Ind_{18}	3836	true	NA	TO	6	17	27.3	15	6	6	*0.37*	**10**
intel007	1307	true	5	**53.5**	NA	NA	TO	NA	NA	NA	NA	TO
intel018	491	true	NA	TO	57	35	695	78	51	33	0.42	**64**
intel019	510	true	NA	TO	52	35	515	96	57	43	0.44	**310**
intel023	358	true	NA	TO	NA	NA	TO	86	53	35	*0.4*	**66**
intel026	492	true	53	47.1	50	35	**21.9**	70	51	34	0.48	27.8

Collaborating with Jasper allowed us to experiment with various real-life industrial designs and properties from various major semiconductor companies.

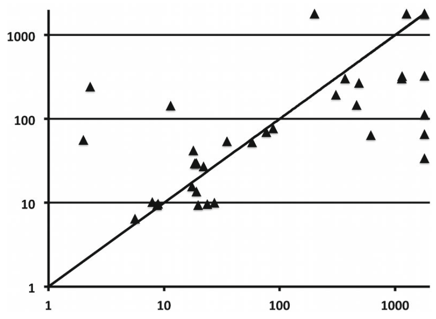
Our implementations use known optimizations for the checked methods (e.g. [2,9]) and are comparable to other optimized implementations available online. For DAR we used some basic procedures to simplify the computed interpolants when possible. Our implementation of DAR is preliminary and can be further optimized.

For the experiments we used 37 real safety properties from real industrial hardware designs. The timeout was set to 1800 seconds and experiments were conducted on systems with Intel Xeon X5660 running at 2.8GHz and 24GB of main memory.

Table 1 shows different parameters for all three algorithms on various industrial examples. *Time* and *convergence depth* are presented for all three, whereas *maximum unrolling* is presented only for ITP and DAR (IC3 does not use unrolling). For DAR we also present ♯GS and GS$_R$ that refer to *global strengthening* (using unrolling) and indicate the number, and ratio, of iterations where *local strengthening* was insufficient.



(a) Runtime DAR vs. IC3.



(b) Runtime DAR vs. ITP.

Fig. 4. Y-axis represents DAR's runtime in seconds. X-axis represents runtime in seconds for the compared algorithm (IC3 or ITP). Points below the diagonal are in favor of DAR.

Examining the results shows that the use of unrolling in DAR is indeed limited and that *local strengthening* plays a major part during verification, with $\mathrm{GS}_R < 0.5$ in most cases, indicating that local strengthening is often sufficient. Moreover, even when unrolling is used, its depth is usually smaller compared to the convergence depth, as indicated by maximum unrolling. Note that the maximum unrolling provides an *upper bound* on the unrolling, and the actual unrolling can be smaller in some global strengthening phases. For falsified properties (counterexample exists) unrolling is necessarily applied up to the length of the counterexample in the last iteration. Yet, in many cases local strengthening is still sufficient in previous iterations.

Another conclusion from the table is that a lower depth of convergence does not necessarily translate to a better runtime. We can see that in many cases, while ITP converges with less computed sets it takes more time than DAR. This is not surprising since the number of computed sets presented for ITP considers only the sets computed in the last bound that was used, disregarding sets from previous bounds. The same can be seen with regards to IC3. While IC3 converges at a lower depth (on some cases), it still does not necessarily perform better. This is mainly due to the different effort invested by each algorithm in the strengthening and addition of a new over-approximated set.

Fig. 4 shows a runtime comparison between DAR and IC3 (Fig. 4a) and ITP (Fig. 4b) on all 37 industrial examples, including those from Table 1. In 19 out of 37 cases, DAR outperforms ITP, and in 25 out of 37 cases it outperforms IC3. In 18 out of 37 cases DAR outperforms both methods. DAR could not solve only 5 cases, whereas ITP and IC3 failed to solve 7 and 12 cases respectively. The overall performance, when summarized, is in favor of DAR with 36% improvement in run time when compared to ITP and 52% improvement when compared to IC3.

Cases where DAR outperforms ITP can be explained by the following factors. First, DAR avoids unrolling when not needed, therefore its SAT calls are simpler. Second, DAR uses over-approximated sets computed in early iterations and strengthens them as needed, while ITP does not re-use sets that were computed for lower bounds and restarts its computation when a spurious counterexample is encountered. Cases where DAR outperforms IC3 are typically when DAR's strengthening is more efficient than IC3's inductive generalization, requiring less computation power at each iteration.

Since DAR relies heavily on interpolants, the cases where DAR performs worse than IC3 are usually those where the interpolants grow large and contain redundancies. This is also true when comparing to ITP. Since DAR computes more interpolants than ITP and also accumulates them, it is more sensitive to the size of the computed interpolants.

We also used the HWMCC'11 benchmark in our experiments. While there are a lot of cases where all methods perform the same, there are also examples where DAR outperforms both IC3 and ITP (some are shown at the bottom of Table 1). The benchmark also includes examples where IC3 or ITP perform better than DAR. The majority of these cases are simple and solved in a few seconds.

6 Conclusions

We present DAR, a complete SAT-based model checking algorithm that uses both *forward* and *backward* interpolants to traverses the state space in a mostly local manner.

The experimental results show that DAR performs well on many industrial designs, and in many cases outperforms the successful ITP and IC3 algorithms. These results are very encouraging, especially since our implementation of DAR can be optimized much further. For example, the local checks applied in the local strengthening phase are independent of each other, which makes DAR most suitable for a parallel implementation.

Our experiments were conducted on hardware designs. However, DAR is not restricted to hardware. It will be interesting to see how it performs on software systems.

Another possible direction for future work refers to an integration of DAR with lazy abstraction [15]. The fact that DAR maintains over-approximations of sets of states reachable from *INIT* or $\neg p$ in *exactly* i steps, rather than in *at most* i steps, enables more flexibility in the choice of abstraction used at each time frame.

References

1. 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)
2. Cabodi, G., Murciano, M., Nocco, S., Quer, S.: Stepping forward with interpolants in unbounded model checking. In: ICCAD, pp. 772–778 (2006)
3. Cabodi, G., Nocco, S., Quer, S.: Mixing Forward and Backward Traversals in Guided-Prioritized BDD-Based Verification. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 471–484. Springer, Heidelberg (2002)
4. Cabodi, G., Nocco, S., Quer, S.: Interpolation sequences revisited. In: DATE (2011)
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. JACM (2003)
6. Clarke, E.C., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
7. Craig, W.: Linear reasoning. A new form of the herbrand-gentzen theorem. J. Symb. Log. 22(3) (1957)
8. D'Silva, V., Purandare, M., Kroening, D.: Approximation Refinement for Interpolation-Based Model Checking. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 68–82. Springer, Heidelberg (2008)
9. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD (2011)

10. Jhala, R., McMillan, K.L.: Interpolant-Based Transition Relation Approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
11. 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)
12. 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)
13. Stangier, C., Sidle, T.: Invariant Checking Combining Forward and Backward Traversal. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 414–429. Springer, Heidelberg (2004)
14. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: FMCAD (2009)
15. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and SAT-based reachability in hardware model checking. In: FMCAD (2012)

An Integrated Specification and Verification Technique for Highly Concurrent Data Structures*

Parosh Aziz Abdulla[1], Frédéric Haziza[1], Lukáš Holík[1,2], Bengt Jonsson[1], and Ahmed Rezine[3]

[1] Uppsala University, Sweden
[2] Brno University of Technology, Czech Republic
[3] Linköping University, Sweden

Abstract. We present a technique for automatically verifying safety properties of concurrent programs, in particular programs which rely on subtle dependencies of local states of different threads, such as lock-free implementations of stacks and queues in an environment without garbage collection. Our technique addresses the joint challenges of infinite-state specifications, an unbounded number of threads, and an unbounded heap managed by explicit memory allocation. Our technique builds on the automata-theoretic approach to model checking, in which a specification is given by an automaton that observes the execution of a program and accepts executions that violate the intended specification. We extend this approach by allowing specifications to be given by a class of infinite-state automata. We show how such automata can be used to specify queues, stacks, and other data structures, by extending a data-independence argument. For verification, we develop a shape analysis, which tracks correlations between pairs of threads, and a novel abstraction to make the analysis practical. We have implemented our method and used it to verify programs, some of which have not been verified by any other automatic method before.

1 Introduction

In this paper, we consider one of the most difficult current challenges in software verification, namely to automate its application to algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state. Such algorithms are of central importance in concurrent programs. They are widely used in libraries, such as the Intel Threading Building Blocks or the java.util.concurrent package, to provide efficient concurrent realizations of simple interface abstractions. They are notoriously difficult to get correct and verify, since they often employ fine-grained synchronization and avoid locking wherever possible. A number of bugs in published algorithms have been reported [10,19]. It is therefore important to develop efficient techniques for verifying conformance to simple abstract specifications of overall functionality, a concurrent implementation of a common data type abstraction, such as a

* Supported in part by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the Czech Science Foundation (project P103/10/0306), and the CENIIT research organization (project 12.04).

queue, should be verified to conform to a simple abstract specification of a (sequential) queue.

We present an integrated technique for specifying and automatically verifying that a concurrent program conforms to an abstract specification of its functionality. Our starting point is the automata-theoretic approach to model checking [30], in which programs are specified by automata that accept precisely those executions that violate the intended specification, and verified by showing that these automata never accept when they are composed with the program. This approach is one of the most successful approaches to automated verification of finite-state programs, but is still insufficiently developed for infinite-state programs. In order to use this approach for our purposes, we must address a number of challenges.

1. The abstract specification is infinite-state, because the implemented data structure may contain an unbounded number of data values from an infinite domain.
2. The program is infinite-state in several dimensions: it (i) consists of an unbounded number of concurrent threads, (ii) uses unbounded dynamically allocated memory, and (iii) the domain of data values is unbounded.
3. The program does not rely on automatic garbage collection, but manages memory explicitly. This requires additional mechanisms to avoid the ABA problem, i.e., that a thread mistakenly confuses an outdated pointer with a valid one.

Each of these challenges requires a significant advancement over current specification and verification techniques.

We cope with challenge 1 by combining two ideas. First, we present a novel technique for specifying programs by a class of automata, called *observers*. They extend automata, as used by [30], by being parameterized on a finite set of variables that assume values from an unbounded domain. This allows to specify properties that should hold for an infinite number of data values. In order to use our observers to specify queues, stacks, etc., where one must "count" the number of copies of a data value that have been inserted but not removed, we must extend the power of observers by a second idea. This is a data independence argument, adapted from Wolper [34], which implies that it is sufficient to consider executions in which any data value is inserted at most once. This allows us to succinctly specify data structures such as queues and stacks, using observers with typically less than 3 variables.

To cope with challenge 2(i), we would like to adapt the successful thread-modular approach [4], which verifies a concurrent program by generating an invariant that correlates the global state with the local state of an arbitrary thread. However, to cope with challenge 3, the generated invariant must be able to express that *at most* one thread accesses some cell on the global heap. Since this cannot be expressed in the thread-modular approach, we therefore extend it to generate invariants that correlate the global state with the local states of an arbitrary *pair* of threads.

To cope with challenge 2(ii) we need to use shape analysis. We adapt a variant of the transitive closure logic by Bingham and Rakamarić [5] for reasoning about heaps with single selectors, to our framework. This formalism tracks reachability properties between pairs of pointer variables, and we adapt it to our analysis, in which pairs of threads are correlated. On top of this, we have developed a novel optimization, based on the observation that it suffices to track the possible relations between each pair of pointer variables separately, analogously to the use of DBMs used in reasoning about timed automata [9]. Finally, we cope with challenge 2(iii) by first observing that data values are compared only by equalities or inequalities, and then employing suitable standard abstractions on the concerned data domains.

We have implemented our technique, and applied it to specify and automatically verify that a number of concurrent programs are linearizable implementation of stacks and queues [16]. This shows that our new contributions result in an integrated technique that addresses the challenges 1 – 3, and can fully automatically verify a range of concurrent implementations of common data structures. In particular, our approach advances the power of automated verification in the following ways.

– We present a direct approach for verifying that a concurrent program is a linearizable implementation of, e.g., a queue, which consists in checking a few small properties of the algorithm, and is thus suitable for automated verification. Previous approaches typically verified linearizability separately from conformance to a simple abstraction, most often using simulation-based arguments, which are harder to automate than simple property-checking.
– We can automatically verify concurrent programs that use explicit memory management. This was previously beyond the reach of automatic methods.

In addition, on examples that have been verified automatically by previous approaches, our implementation is in many cases significantly faster.

Overview. We give an overview of how our technique can be used to show that a concurrent program is a linearizable implementation of a data structure. As described in Section 2, we consider concurrent programs consisting of an arbitrary number of sequential threads that access shared global variables and a shared heap using a finite set of methods. Linearizability provides the illusion that each method invocation takes effect instantaneously at some point (called the linearization point) between method invocation and return [16]. In Section 3, we show how to specify this correctness condition by first instrumenting each method to generate a so-called abstract event whenever a linearization point is passed. We also introduce *observers*, and show how to use them for specifying properties of sequences of abstract events. In Section 4, we introduce the data independence argument that allows observers to specify queues, stacks, and other unbounded data structures. In Section 5, we describe our analysis for checking that the cross-product of the program and the observer cannot reach an accepting location of the observer. The analysis is based on a shape analysis, which generates an invariant that correlates the global state with the local states of an arbitrary pair of threads. We also introduce our optimization which tracks the possible relations between each pair of pointer variables separately. We report on experimental results in Section 6. Section 7 contains conclusions and directions for future work.

Related work. Much previous work on verification of concurrent programs has concerned the detection of generic concurrency problems, such as race conditions, atomicity violations, or deadlocks [14,22,23]. Verification of conformance to a simple abstract specification has been performed using refinement techniques, which establish simulation relations between the implementation and specification, using partly manual techniques [11,8,12,33].

Amit et al [3] verify linearizability by verifying conformance to an abstract specification, which is the same as the implementation, but restricted to serialized executions. They build a specialized abstract domain that correlates the state (including the heap cells) of a concrete thread and the state of the serialized version, and a sequential reference data structure. The approach can handle a bounded number of threads. Berdine et al [4] generalize the approach to an unbounded number of threads by making the shape analysis thread-modular. In our approach, we need not keep track of heaps emanating from sequential reference executions, and so we can use a simpler shape analysis. Plain thread-modular analysis is also not powerful enough to analyze e.g. algorithms with explicit memory management. [4] thus improves the precision by correlating local states of different threads. This causes however a severe state-space explosion which limits the applicability of the method.

Vafeiadis [27] formulates the specification using an unbounded sequence of data values that represent, e.g., a queue or a stack. He verifies conformance using a specialized abstraction to track values in the queue and correlate them with values in the implementation. Like [25], our technique for handling values in queues need only consider a small number of data values (not an unbounded one), for which it is sufficient to track equalities. The approach is extended in [28] to automatically infer the position of linearization points: these have to be supplied in our approach.

Our use of data variables in observers for specifying properties that hold for all data values in some domain is related in spirit to the identification of arbitrary but fixed objects or resources by Emmi et al. [13] and Kidd et al. [18]. In the framework of regular model checking, universally quantified temporal logic properties can be compiled into automata with data variables that are assigned arbitrary initial values [1].

Segalov et al. [24] continue the work of [4] by also considering an analysis that keeps track of correlations between threads. They strive to counter the state-space explosion that [4] suffers from, and propose optimizations that are based on the assumption that inter-process relationships that need to be recorded are relatively loose, allowing a rather crude abstraction over the state of one of the correlated threads. These optimizations do not work well when thread correlations are tight. Our experimental evaluation in Section 6 shows that our optimizations of the thread correlation approach achieve significantly better analysis times than [24].

There are several works that apply different verification techniques to programs with a bounded number of threads, including the use of TVLA [35]. Several approaches produce decidability results under limited conditions [7], or techniques based on non-exhaustive testing [6] or state-space exploration [32] for a bounded number of threads.

2 Programs

We consider systems consisting of an arbitrary number of concurrently executing threads. Each thread may at any time invoke one of a finite set of methods. Each method declares local variables (including the input parameters of the method) and a method body. In this paper, we assume that variables are either pointer variables (to heap cells), or data variables (assuming values from an unbounded or infinite domain, which will be denoted by

$\mathbb{D}$). The body is built in the standard way from atomic commands using standard control flow constructs (sequential composition, selection, and loop constructs). Method execution is terminated by executing a `return` command, which may return a value. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method. We assume that the global variables and the heap are initialized by an initialization method, which is executed once at the beginning of program execution.

Atomic commands include assignments between data variables, pointer variables, or fields of cells pointed to by a pointer variable. The command **new** `node()` allocates a new structure of type `node` on the heap, and returns a reference to it. The cell is deallocated by the command **free**. The compare-and-swap command `CAS(&a,b,c)` atomically compares the values of `a` and `b`. If equal, it assigns the value of `a` to `c` and returns `TRUE`, otherwise, it leaves `a` unchanged and returns `FALSE`.

As an example, Figure 1 shows a version of the concurrent queue by Michael and Scott [20]. The program represents a queue as a linked list from the node pointed to by `Head` to a node that is either pointed by `Tail` or by `Tail`'s successor. The global variable `Head` always points to a dummy cell whose successor, if any, stores the head of the queue. In the absence of garbage collection, the program must handle the ABA problem where a thread mistakenly assumes that a globally accessible pointer has not been changed since it previously accessed that pointer. Each pointer is therefore equipped with an additional `age` field, which is incremented whenever the pointer is assigned a new value.

The queue can be accessed by an arbitrary number of threads, either by an enqueue method `enq`(`d`), which inserts a cell containing the data value `d` at the tail, or by a dequeue method `deq`(`d`) which returns `empty` if the queue is empty, and otherwise advances `Head`, deallocates the previous dummy cell and returns the data value stored in the new dummy cell. The algorithm uses the atomic compare-and-swap (CAS) operation. For example, the command `CAS(&Head, head, ⟨next.ptr,head.age+1⟩)` at line 29 of the `deq` method checks whether the extended pointer `Head` equals the extended pointer `head` (meaning that both fields must agree). If not, it returns `FALSE`. Otherwise it returns `TRUE` after assigning `⟨next.ptr,head.age+1⟩` to `Head`.

3 Specification by Observers

To specify a correctness property, we instrument each method to generate abstract events. An *abstract event* is a term of the form $l(d_1, \ldots, d_n)$ where l is an event type, taken from a finite set of event types, and $d_1, \ldots, d_n$ are data values in $\mathbb{D}$. To specify linearizability, the abstract event $l(d_1, \ldots, d_n)$ generated by a method should be such that l is the name of the method, and $d_1, \ldots, d_n$ is the sequence of actual parameters and return values in the current invocation of the method. This can be established using standard sequential verification techniques.

We illustrate how to instrument the program of Figure 1 in order to specify that it is a linearizable implementation of a queue. The linearization points ● are at line 9, 21 and 29. For instance, line 9 of the `enq` method called with data value `d` is instrumented to generate the abstract event `enq`(`d`) when the `CAS` command succeeds; no abstract event

```
   void initialize() {                                INIT
     node* n := new node();
     n→next.ptr := NULL;
     Head.ptr := n;
     Tail.ptr := n;
   }

0  void enq(data d){                                  ENQ
1    node* n := new node();
2    n→val := d;
3    n→next.ptr := NULL;
4    while(TRUE){
5      pointer_t tail := Tail;
6      pointer_t next := tail.ptr→next;
7      if(tail = Tail)
8        if(next.ptr = NULL)
9          if(CAS(&tail.ptr→next, next,  ●
10             ⟨n,next.age+1⟩))
11           break;
12       else
13         CAS(&Tail,tail,⟨next.ptr, tail.age+1⟩);
14   }
15   CAS(&Tail, tail, ⟨n, tail.age+1⟩);
16 }
```

```
   struct node {data val, pointer_t next}
   struct pointer_t {node* ptr, int age}

   pointer_t Head, Tail;

17 data deq(){                                        DEQ
18   while(TRUE){
19     pointer_t head := Head;
20     pointer_t tail := Tail;
21     pointer_t next := head.ptr→next;  ●
22     if(head = Head)
23       if(head.ptr = tail.ptr)
24         if(next.ptr = NULL)
25            return empty;
26         CAS(&Tail, tail, ⟨next.ptr, tail.age+1⟩);
27       else
28         data result := next.ptr→val;
29         if(CAS(&Head, head,  ●
30              ⟨next.ptr,head.age+1⟩))
31            break;
32   }
33   free(head.ptr);
34   return result;
35 }
```

Fig. 1. Michael & Scott's non-blocking queue [20]

is generated when the `CAS` fails. Generation of abstract events can be conditional. For instance, line 21 of the `deq` method is instrumented to generate `deq(empty)` when the value assigned to `next` satisfies `next.ptr = NULL` (i.e., it will cause the method to return `empty` at line 25).

Each execution of the instrumented program will generate a sequence of abstract events called a *trace*. A *correctness property* (or simply a *property*) is a set of traces. We say that an instrumented program *satisfies* a property if each trace of the program is in the property. In contrast to the classical (finite-state) automata-theoretic approach [30], we specify properties by *infinite-state* automata, called *observers*. An observer has a finite set of control locations, and a finite set of data variables that range over potentially infinite domains. It observes the trace and can reach an accepting control location if the trace is not in the property.

Formally, let a *parameterized event* be a term of the form $l(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are formal parameters. We will write $\overline{p}$ for $p_1, \ldots, p_n$, and $\overline{d}$ for $d_1, \ldots, d_n$. An *observer* consists of a finite set of *observer locations*, one of which is *initial* and some of which are *accepting*, a finite set of *observer variables*, and a finite set of *transitions*. Each transition is of form $s \xrightarrow{l(\overline{p});g} s'$ where s, s' are observer locations, $l(\overline{p})$ is a parameterized event, and the guard g is a Boolean combination of equalities over formal parameters $\overline{p}$, and observer variables. Intuitively, it denotes that the observer can move from location s to location s' when an abstract event of form $l(\overline{d})$ is generated such that $g[\overline{d}/\overline{p}]$ is true. Note that the values of observer variables are not updated in a transition. An *observer configuration* is a pair $\langle s, \vartheta \rangle$, where s is an observer location, and ϑ maps each observer variable to a value in the data domain $\mathbb{D}$. The configuration is initial if s is initial; thus the variables can assume any initial values. An *observer step* is a triple $\langle s, \vartheta \rangle \xrightarrow{l(\overline{d})} \langle s', \vartheta \rangle$ such that there is a transition $s \xrightarrow{l(\overline{p});g} s'$ for which $g[\overline{d}/\overline{p}]$

is true. A *run* of the observer on a trace $\sigma = l_1(\overline{d}_1)l_2(\overline{d}_2)\cdots l_n(\overline{d}_n)$ is a sequence of observer steps $\langle s_0, \vartheta\rangle \xrightarrow{\mathtt{l}_1(\overline{d}_1)} \cdots \xrightarrow{\mathtt{l}_n(\overline{d}_n)} \langle s_n, \vartheta\rangle$ where s_0 is the initial observer location. The run is *accepting* if s_n is accepting. A trace σ is *accepted* by an observer $\mathcal{A}$ if $\mathcal{A}$ has an accepting run on σ. The property specified by $\mathcal{A}$ is the set of traces that are not accepted by $\mathcal{A}$.

Since the data variables can assume arbitrary initial values, observers can specify properties that are universally quantified over all data values. If a trace violates such a property for some data values, the observer can non-deterministically choose these as initial values of its variables, and thereafter detect the violation when observing the trace. Several data structures can be specified by a collection of properties, each of which is represented by an observer. For instance, a set can be specified by a collection of properties, one of which is that a data value can be deleted, only if it has been previously inserted. The observer in Figure 2 specifies this property: it accepts executions in which for some data value d, a $\mathtt{delete}(d)$-event is observed even though no earlier $\mathtt{insert}(d)$-event has been observed.

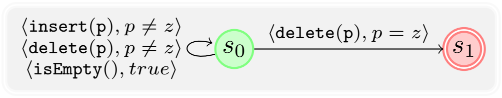


Fig. 2. An observer for checking that no data value can be deleted if it has not been first inserted. The variable z is an observer variable.

4 Data Independence

In the previous section, we showed how observers can specify some data structures, such as sets, in a straight-forward way. However, in order to specify some other data structures, including queues and stacks, for which one must be able to "count" the number of copies of a data value that have been inserted but not removed, we must additionally employ an extension of a data independence argument, originating from Wolper [34], as follows.

The argument assumes that for each trace, there is a fixed subset of all occurrences of data values in the trace, called the set of *input occurrences*. Formally, this subset can be arbitrary, but to make the argument work, input occurrences should typically be the data values that are provided as actual parameters of method invocations. Thus, in the program of Figure 1, the input occurrences are the parameters of $\mathtt{enq}(\mathtt{d})$ events, whereas parameters of $\mathtt{deq}(\mathtt{d})$ events are *not* input occurrences, since they are provided as return values.

Let us introduce some definitions. A trace is *differentiated* if all its input occurrences are pairwise different. A *renaming* is any function $f : \mathbb{D} \mapsto \mathbb{D}$ on the domain of data values. A renaming f can be applied to trace σ, resulting in the trace $f(\sigma)$, where each data value $\mathtt{d}$ in σ has been replaced by $f(\mathtt{d})$. A set Σ of traces is *data independent* if for any trace $\sigma \in \Sigma$ the following two conditions hold:

- $f(\sigma) \in \Sigma$ for any renaming f, and
- there exists a differentiated trace $\sigma_d \in \Sigma$ with $f(\sigma_d) = \sigma$ for some renaming f.

We say that a program is *data independent* if the set of its traces is data independent. A program, like the one in Figure 1, can typically be shown to be data independent by a simple syntactic analysis that checks that data values are not manipulated or tested, but only copied. In a similar manner, a correctness property is *data independent* if the set of traces that it specifies is data independent. From these definitions, the following theorem follows directly.

Theorem 1. *A data independent program satisfies a data independent property iff its differentiated traces satisfy the property.* □

Thus, when checking that a data independent program satisfies a data independent property, it suffices to check that all differentiated traces of the program belong to the property. Hence, an observer for a data independent property need only accept the differentiated traces that violate the property. It should not accept any (differentiated or non-differentiated) trace that satisfies it.

Note that the set of traces of a set is *not* data independent, e.g., since it contains a trace where two different data values are inserted, but *not* its renaming which inserts the same data value twice. This is not a problem, since the set of *all* traces of a set can be specified by observers, without using a data independence argument.

The key observation is now that the differentiated traces of queues and stacks can be specified succinctly by observers with a small number of variables. In the case of a FIFO queue, its differentiated traces are precisely those that satisfy the following four properties for all data values d_1 and d_2.

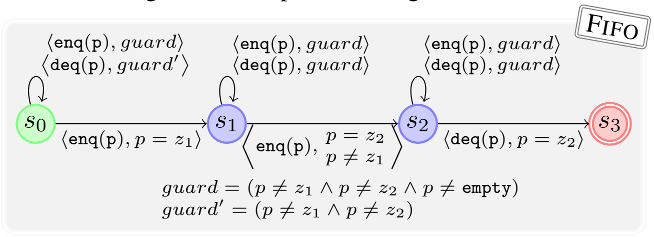

FIFO
s_0
s_1
s_2
s_3
$\langle \texttt{enq(p)}, guard \rangle$
$\langle \texttt{deq(p)}, guard' \rangle$
$\langle \texttt{enq(p)}, guard \rangle$
$\langle \texttt{deq(p)}, guard \rangle$
$\langle \texttt{enq(p)}, guard \rangle$
$\langle \texttt{deq(p)}, guard \rangle$
$\langle \texttt{enq(p)}, p = z_1 \rangle$
$\left\langle \texttt{enq(p)}, \begin{array}{l} p = z_2 \\ p \neq z_1 \end{array} \right\rangle$
$\langle \texttt{deq(p)}, p = z_2 \rangle$
$guard = (p \neq z_1 \land p \neq z_2 \land p \neq \texttt{empty})$
$guard' = (p \neq z_1 \land p \neq z_2)$

Fig. 3. An observer to check that FIFO ordering is respected. All unmatched abstract events, for example $\langle \texttt{deq(p)}, p = z_1 \rangle$ at location s_1, send the observer to a sink state.

NO CREATION: d_1 must not be dequeued before it is enqueued
NO DUPLICATION: d_1 must not be dequeued twice,
NO LOSS: `empty` must not be returned if d_1 has been enqueued but not dequeued,
FIFO: d_2 must not be dequeued before d_1 if it is enqueued after d_1 is enqueued.

Each such property can be specified by an observer with one or two variables. If the property is violated by some specific data values d_1 and d_2, then there is some run of the observer, in which the initial values of the variables are d_1 and d_2, which leads to an accepting state. Figure 3 shows an observer for the FIFO property.

We can also provide an analogous characterization of the differentiated traces of a stack.

5 Verification by Shape Analysis

To verify that no trace of the program is accepted by an observer, we form, as in the automata-theoretic approach [30], the cross-product of the program and the observer, synchronizing on abstract events, and check that this cross-product cannot reach a configuration where the observer is in an accepting state.

The analysis needs to deal with the challenges of an unbounded data domain, an unbounded number of concurrently executing threads, an unbounded heap, and an explicit memory management. As indicated in Section 1, the explicit memory management implies that the assertions generated by our analysis must be able to track correlations between pairs of threads. We present our shape analysis in two steps. We first describe a symbolic encoding of the configurations of the program and then present the verification procedure.

5.1 Representation of Symbolic Encodings

A symbolic encoding characterizes *all* the reachable configurations of the program from the point of view of two distinct executing threads. This is done by recording the relationships of the local configurations of the two threads with each other, the relationships of the local variables of a thread with global variables, the observer configuration, and the assertions about the heap. It is a combination of several layers of conjunctions and disjunctions. In this section, let us fix two thread identifiers i_1 and i_2 and let us first introduce some necessary definitions in a bottom-up manner.

Cell terms. Let a *cell term* be one of the following: (i) a global pointer variable y, which denotes the cell pointed to by the global variable y, (ii) a term of the form $x[i_j]$ (where $j = 1$ or $j = 2$) for a local pointer variable x of thread i_j, which denotes the cell pointed to by the thread-i_j-local-copy of x, (iii) a special term `NULL`, `UNDEF`, or `FREE`, or (iv) a cell variable, which denotes a cell whose data value is equal to the current value of an observer variable. (Note that the value of an observer variable is fixed during a run of the observer). The latter allows us to keep track of the data in the heap cells, even in the case where a heap cell is not denoted by any pointer variable (in order to verify, e.g., the FIFO property of a queue). We use $CT(i_1, i_2)$ to denote the set of all cell terms (of thread i_1 and i_2).

Atomic heap constraint. In order to obtain an efficient and practical analysis, which does not lead to a severe explosion of formulas, we have developed a novel representation, adapted from the transitive closure logic of [5]. The representation is motivated by the observation that relationships between pairs of pointer variables are typically independent. The key aspect of the representation is that it is sufficient to consider only pairs of variables rather than correlating all variables. An atomic heap constraint is of one of the following forms (where t_1 and t_2 are two cell terms):

- $t_1 = t_2$: the cell terms t_1 and t_2 denote the same cell,
- $t_1 \mapsto t_2$: the `next` field of the cell denoted by t_1 denotes the cell denoted by t_2,
- $t_1 \dashrightarrow t_2$: the cell denoted by t_2 can be reached by following a chain of two or more `next` fields from the cell denoted by t_1,
- $t_1 \bowtie t_2$: none of $t_1 = t_2$, $t_1 \mapsto t_2$, $t_2 \mapsto t_1$, $t_1 \dashrightarrow t_2$, or $t_2 \dashrightarrow t_1$ is true.

We use *Pred* to denote the set $\{=, \mapsto, \hookleftarrow, \dashrightarrow, \dashleftarrow, \bowtie\}$ of all shape relational symbols. We let $t =$ `NULL` denote that t is null, $t \mapsto$ `UNDEF` denote that t is undefined, and $t \mapsto$ `FREE` denote that t is unallocated.

Joined shape constraint. A joined shape constraint, for thread i_1 and i_2, denoted as $M(i_1, i_2)$, is a (typically large) conjunction $\bigwedge_{t_1,t_2 \in CT(i_1,i_2)} \pi[t_1, t_2]$ where $\pi[t_1, t_2]$ is a non-empty disjunction of atomic heap constraints. Intuitively, it is a matrix representing the heap parts accessible by the two threads (along with the cell data). Such a representation can be (exponentially) more concise than using a large disjunction of conjunctions of atomic heap constraints, at the cost of some loss of precision.

We say that a joined shape constraint $M(i_1, i_2)$ is *saturated* if for all terms x, y, and z in $CT(i_1, i_2)$, every atomic heap constraint from the disjunction $\pi[x, z]$ implies the heap constraints that one can derive from those found in $\pi[x, x]$, $\pi[x, y]$, $\pi[y, y]$, $\pi[y, z]$, and $\pi[z, z]$. Any joined shape constraint can be saturated by a straightforward fixpoint procedure, analogous to [5] or the one for DBMs [9]. For instance, let $\pi[x, z]$ be $x \mapsto z$ and $\pi[y, z]$ be $y \mapsto z \vee y \dashrightarrow z$ and let $\pi[x, x]$ and $\pi[y, y]$ admit only equality (there is no loop involving x or y). Then $\pi[x, y]$ can contain the disjuncts $x = y$, $x \bowtie y$, which are consistent with $x \mapsto z$ and $y \mapsto z$. It can also contain $x \leftarrow\!\shortmid y$, $x \dashleftarrow y$, and $x \bowtie y$, that are consistent with $x \mapsto z$ and $y \dashrightarrow z$. In short, x cannot reach y, thus when saturating, we remove $x \mapsto y$ and $x \dashrightarrow y$ from $\pi[x, y]$.

Symbolic Encoding. We can now define formally a symbolic encoding over two threads. A symbolic encoding is a disjunction $\Theta[i_1, i_2]$ of formulas of the form $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ where $\sigma[i_1, i_2]$ is a *control formula* and $\phi[i_1, i_2]$ is a *shape formula*.

A *control formula* $\sigma[i_1, i_2]$ contains (i) the current control location of threads i_1 and i_2, and the observer, and (ii) a conjunction encompassing the relations between the `age` fields of any pair of terms. For instance, when analyzing the program in Figure 1, this conjunction includes among others, for a thread i, `head[i].age`$\simeq$`Head.age` and `tail[i].ptr`$\rightarrow$`next.age`$\simeq$`next[i].age`, where $\simeq \in \{<, =, >\}$.

A *shape formula* $\phi[i_1, i_2]$ is a joined shape constraint conjoined with a formula $\psi[v_1, \ldots, v_m, z_1, \ldots, z_n]$ which links the cell variables $v_1, \ldots, v_m$ with the observer variables $z_1, \ldots, z_n$ that are used to keep track of heap cells with values equal to the observer variables. Formally, $\phi[i_1, i_2]$ is a formula of the form

$$\exists v_1, \ldots, v_m. \, [\psi[v_1, \ldots, v_m, z_1, \ldots, z_n] \;\wedge\; M(i_1, i_2)]$$

5.2 Verification Procedure

We compute an invariant of the program of the form $\forall i_1, i_2.\ (i_1 \neq i_2 \Rightarrow \Theta[i_1, i_2])$ which characterizes the configurations of the program from the point of view of two distinct executing threads i_1 and i_2. We obtain the invariant by a standard fixpoint procedure, starting from a formula that characterizes the set of initial configurations of the program. For two distinct threads i_1 and i_2, and for each control formula $\sigma[i_1, i_2]$, our analysis will generate one shape formula $\phi[i_1, i_2]$.

The fixpoint analysis performs a postcondition computation that results in a set of possible successor combinations of control and shape formulas. The new shape formulas of which the control formula already appears in the original $\Theta[i_1, i_2]$ will be used to weaken the corresponding old shape formula. Otherwise, if the control state is new, a new disjunct is added to $\Theta[i_1, i_2]$.

For two threads i_1 and i_2, we must consider two scenarios: either i_1 or i_2 performs a step, or some other (interfering) thread i_3, (distinct from i_1 and i_2), performs a step.

Postcondition computation. In the first scenario, where one of the threads i_1 or i_2 performs a step, we can compute the postcondition of $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ as follows. $\sigma[i_1, i_2]$ is first updated to a new control state $\sigma'[i_1, i_2]$ in the standard way (by updating the possible values of control locations and observer state). $\phi[i_1, i_2]$ is then updated to $\phi'[i_1, i_2]$ by updating each conjunct $\pi[t_1, t_2]$ according to the particular program statement that the thread is performing. In general, we (i) remove all disjuncts that must be falsified by the step (this may require splitting the formula into several stronger formulas whenever the falsification might be ambiguous), (ii) add all disjuncts that may become true by the step, (iii) saturate the result.

Consider for instance the program statement `x:=y.next`. Since only the value of x is changing, the transformer updates only conjuncts $\pi[t, x]$ and $\pi[x, t]$ where $t \in CT(i_1, i_2)$. All assertions about x are reset by setting every conjunct $\pi[x, t]$ and $\pi[t, x]$ to *Pred*, for all $t \in CT(i_1, i_2)$. (The disjunction over all elements of *Pred* is the assertion $true$). We then set $\pi[x, y]$ to $x \hookleftarrow y$, $\pi[y, x]$ to $y \mapsto x$ and derive all predicates that may follow by transitivity. Finally, we saturate the formula. It prunes the (newly added) predicates that are inconsistent with the rest of the shape formula.

For `x.next:=y`, it is important to know the reachabilities that depend on the pointer x.`next`. The representation might potentially contain imprecision (it might for instance state that, for a term t, $\pi[t, x]$ contains $t \dashleftarrow x$ and $t \dashrightarrow x$, even if we know, via a simpler analysis, that no cycles are generated). Hence, we first split the formula into stronger formulas in such a way that we disambiguate the part of the reachability relation involving x. On each resulting formula, we then remove reachability predicates between cell terms that depend on x.`next` (e.g., we remove $u \dashrightarrow v$ if $u \dashrightarrow x$ and $x \dashrightarrow v$). We then set $\pi[x, y]$ to $x \mapsto y$ and derive all predicates that may follow by transitivity (e.g., if $u \dashrightarrow x$ and $y \dashrightarrow v$, we add $u \dashrightarrow v$), and we saturate the result.

Interference. In the case where we need to account for possible interference on the formula $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ by another thread, (distinct from i_1 or i_2), we proceed as follows. We (i) extend the formula with the interfering thread, (ii) compute a postcondition as described in the first scenario and (iii) project away the interfering thread.

Step (i) combines a given formula $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ with the information of an extra thread i_3. In a similar manner to [2], the resulting formula is of the form $(\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3])$ such that any projection to two threads is a formula compatible with some disjunct of $\Theta[i_1, i_2]$. To generate all such formulas involving three threads, we must, besides $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ itself, consider all pairs of disjuncts $(\sigma_\bullet[i_2, i_3] \wedge \phi_\bullet[i_2, i_3])$ and $(\sigma_\circ[i_1, i_3] \wedge \phi_\circ[i_1, i_3])$, such that $\sigma[i_1, i_2] \wedge \sigma_\bullet[i_2, i_3] \wedge \sigma_\circ[i_1, i_3]$ is consistent. In this case, we generate the formula $\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3]$ where $\sigma[i_1, i_2, i_3] \equiv \sigma[i_1, i_2] \wedge \sigma_\bullet[i_2, i_3] \wedge \sigma_\circ[i_1, i_3]$ and $\phi[i_1, i_2, i_3] \equiv \phi[i_1, i_2] \wedge \phi_\bullet[i_2, i_3] \wedge \phi_\circ[i_1, i_3]$. We then saturate $\phi[i_1, i_2, i_3]$ (in the same way as for joined shape formulas over two threads). For each statement S of thread i_3 that can be executed when $\sigma[i_1, i_2, i_3]$ holds, we compute (step ii) its postcondition $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$. Finally (step iii), $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$ is projected back onto $\sigma'[i_1, i_2] \wedge \phi'[i_1, i_2]$ by removing all information about the variables of thread i_3.

Since the domain of control formulas and the domain of shape formulas over a fixed number of cell terms are finite, the abstract domain of formulas $\forall i_1, i_2.\ (i_1 \neq i_2 \Rightarrow \Theta[i_1, i_2])$ is finite as well. The iteration of postcondition computation is thus guaranteed to terminate.

6 Experimental Results

We have implemented a prototype in OCaml and used it to automatically establish the conformance of concurrent data-structures (including lock-free and lock-based stacks, queues and priority queues) to their operational specification (implying their linearizability). Our analysis also implicitly checks for standard shape-related errors such as null/undefined pointer dereferencing (taking into account the known dangling pointers' dereferences [21]), double-free, or presence of cycles.

Some of the examples are verified in the absence of garbage collection, in particular, the lock-free versions of Treiber's [26] stack and Michael&Scott's queue (see Figure 1). We hereafter refer to them as Treiber's stack and M&S's queue, and garbage collection as GC. The verification of these examples is extensively demanding as it requires to correlate the possible states of the threads with high precision. We are not aware of any other method capable of verifying high level functionality of these benchmarks. We ran the experiments on a 3.5 GHz processor with 8GB memory. We report, in Table 1, the running times (in seconds) and the final number of joined shape constraints generated ($|C|$, reduced by symmetry).

Table 1. Experimental Results. (stack+ (resp. queue+) is an observer encompassing the loss, creation, duplication and lifo (resp. fifo) observers)

	Conformance			Safety only	
Data-structure	**Observers**	**Time**	$\lvert C\rvert$	**Time**	$\lvert C\rvert$
Coarse Stack	stack+	0.02s	436	0.01s	102
Coarse Stack, no GC	stack+	0.07s	569	0.01s	130
Coarse Queue	queue+	0.04s	673	0.01s	196
Coarse Queue, no GC	queue+	0.48s	1819	0.10s	440
Two-Locks Queue[20]	queue+	0.08s	1830	0.02s	488
Two-Locks Queue, no GC	queue+	0.73s	3460	0.13s	784
		vs 47s in [4]		*vs* 6162s/304s in [35]	
Coarse Priority Queue (Buckets)	prio	0.24s	1242	0.07s	526
Coarse Priority Queue (List-based)	prio	0.04s	499	0.01s	211
Bucket locks Priority Queue	prio	0.22s	1116	0.05s	372
Treiber's lock-free stack[26]	stack+	0.23s	714	0.01s	78
		vs 0.09s in [29]			
Treiber's lock-free stack, no GC	stack+	2.28s	1535	0.10s	190
		vs 53s in [4]			
M&S's lock-free queue[20]	queue+	3.31s	3476	0.44s	594
		vs 3.36s in [29]			
M&S's lock-free queue, no GC	queue+	550s	53320	25s	6410
		vs o.o.m. in [4]		*vs* 727s/309s in [35]	

Table 2. Introducing intentional bugs: The analysis is sound and the programs are not verified

Data-structure	Modification	Observer	Output	Time
Treiber's stack	none	fifo	bad trace	0.07s
Treiber's stack, no GC	none	fifo	bad trace	6.19s
M&S's queue	none	lifo	bad trace	1.26s
Two-locks queue	bad commit point	fifo	bad trace	0.02s
M&S's queue	bad commit point	loss	bad trace	0.51s
Treiber's stack	omitting data	lifo	bad trace	0.02s
Treiber's stack, no GC	discard ages	loss	bad trace	0.42s
Treiber's stack, no GC	discard ages	loss	cycle creation	0.01s
M&S's queue, no GC	discard ages	loss	bad trace	272s
M&S's queue, no GC	discard ages	loss	dereferencing null	0.01s
M&S's queue	swapped assignments	memo	dereferencing null	0.01s

In addition to establishing correctness of the original versions of the benchmark programs, we also stressed our tool with few examples in which we intentionally inserted bugs (cf. Table 2). As expected, the tool did not establish correctness of these erroneous programs since the approach is sound. For example, we tested whether stacks (resp. queues) implementations can exhibit fifo (resp. lifo) traces, we tested whether values can be lost (loss observer), or memory errors can be triggered (memo observer accepts on memory errors made visible), we moved linearization points to wrong positions, and we tested a program which stores wrong values of inserted data. In all these cases, the analysis correctly reported traces that violated the concerned safety property. Finally, we ran the data structure implementations without garbage collection discarding the `age` counters and our (precise) analysis produced as expected a trace involving the ABA problem [17].

We also include a succinct comparison with related work. Although it is often unfair to compare approaches solely based on running times of different tools, we believe that such a comparison can give an idea of the efficiency of the involved approaches. Our running times on the versions of Treiber's stack and M&S's queue that assume GC are comparable with the results of [29]. However, the verification of versions that do not assume GC is, to the best of our knowledge, beyond the reach of [29] (since it does not correlate states of different threads). [24] verifies linearizability of concurrent implementations of sets, e.g., a lock-free `CAS`-based set [31] (verified in 2975s) of a comparable complexity to M&S's queue without GC (550s with our prototype). Basic memory safety of M&S's queue and two-locks queue [20] without GC was also verified in [35], but only for a scenario where all threads are either dequeuing or enqueuing. The verification took 727s and 309s for M&S's queue and 6162s and 304s for the two-locks queue. Our verification analysis produced the same result significantly faster, even allowing any thread to non deterministically decide to either enqueue or dequeue. In [4], linearizability of the Treibers's stack (resp. two-locks queue [20]) is verified in 53s (resp. 47s). We achieve the same result in less than 3 seconds. Finally, a variant of M&S's queue without GC could not be verified in [4] due to lack of memory.

7 Conclusions and Future Work

We have presented a technique for automated verification of temporal properties of concurrent programs, which can handle the challenges of infinite-state specifications, an unbounded number of threads, and an unbounded heap managed by explicit memory allocation. We showed how such a technique can be based naturally on the automata-theoretic approach to verification, by nontrivial combinations and extensions that handle unbounded data domains, unbounded number of threads, and heaps of arbitrary size. The result is a simple and direct method for verifying correctness of concurrent programs. The power of our specification formalism is enhanced by showing how the data-independence argument by Wolper [34] can be introduced into standard program analysis. Our method can be parameterized by different shape analyses. Although we concentrate on heaps with single selectors in the current paper, we expect that our method can be adapted to deal with multiple selectors, by integrating recent approaches such as [15]. Morever, our experminatation deals with the specification of stacks and queues. Other data structures, such as deques, can be handled in an analogous way.

References

1. Abdulla, P., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). STTT 14(2), 223–241 (2012)
2. Abdulla, P.A., Haziza, F., Holík, L.: All for the Price of Few (Parameterized Verification through View Abstraction). In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 476–495. Springer, Heidelberg (2013)
3. Amit, D., Rinetzky, N., Reps, T., 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)
4. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread Quantification for Concurrent Shape Analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)
5. Bingham, J., Rakamarić, Z.: A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 207–221. Springer, Heidelberg (2006)
6. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proc. of PLDI 2010, pp. 330–340. ACM (2010)
7. 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)
8. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 475–488. Springer, Heidelberg (2006)
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. Doherty, S., Detlefs, D., Groves, L., Flood, C., Luchangco, V., Martin, P., Moir, M., Shavit, N., Steele Jr., G.: Dcas is not a silver bullet for nonblocking algorithm design. In: Proc. of SPAA 2004, pp. 216–224. ACM (2004)
11. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal Verification of a Practical Lock-Free Queue Algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)

12. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying Linearizability Proofs with Reduction and Abstraction. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 296–311. Springer, Heidelberg (2010)
13. Emmi, M., Jhala, R., Kohler, E., Majumdar, R.: Verifying Reference Counting Implementations. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 352–367. Springer, Heidelberg (2009)
14. Flanagan, C., Freund, S.: Atomizer: A dynamic atomicity checker for multithreaded programs. Science of Computer Programming 71(2), 89–109 (2008)
15. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. Formal Methods in System Design, 1–24 (2012)
16. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
17. IBM. System/370 principles of operation (1983)
18. Kidd, N., Reps, T., Dolby, J., Vaziri, M.: Finding concurrency-related bugs using random isolation. STTT 13(6), 495–518 (2011)
19. Michael, M., Scott, M.: Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Rochester, NY, USA (1995)
20. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on PoDC, pp. 267–275 (1996)
21. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: Proc. 21st Annual Symp. on PoDC, pp. 21–30 (2002)
22. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proc. of PLDI 2006, pp. 308–319. ACM (2006)
23. Naik, M., Park, C.-S., Sen, K., Gay, D.: Effective static deadlock detection. In: Proc. of ICSE, pp. 386–396. IEEE (2009)
24. Segalov, M., Lev-Ami, T., Manevich, R., Ganesan, R., Sagiv, M.: Abstract Transformers for Thread Correlation Analysis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 30–46. Springer, Heidelberg (2009)
25. Shacham, O.: Verifying Atomicity of Composed Concurrent Operations. PhD thesis, Department of Computer Science, Tel Aviv University (2012)
26. Treiber, R.: Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr. (1986)
27. Vafeiadis, V.: Shape-Value Abstraction for Verifying Linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)
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.: RGSep Action Inference. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 345–361. Springer, Heidelberg (2010)
30. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. of LICS 1986, pp. 332–344 (June 1986)
31. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Proc. of PLDI 2008, pp. 125–135. ACM (2008)
32. Vechev, M., Yahav, E., Yorsh, G.: Experience with Model Checking Linearizability. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009)
33. Wang, L., Stoller, S.: Static analysis of atomicity for programs with non-blocking synchronization. In: Proc. of PPOPP 2005, pp. 61–71. ACM (2005)
34. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic (extended abstract). In: Proc. of POPL 1986, pp. 184–193 (1986)
35. Yahav, E., Sagiv, S.: Automatically verifying concurrent queue algorithms. Electr. Notes Theor. Comput. Sci. 89(3) (2003)

A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems*

Alexander Linden and Pierre Wolper

Institut Montefiore, B28
Université de Liège
B-4000 Liège, Belgium
{linden,pw}@montefiore.ulg.ac.be

Abstract. This paper addresses the problem of verifying and correcting programs when they are moved from a sequential consistency execution environment to a relaxed memory context. Specifically, it considers the PSO (Partial Store Order) memory model, which corresponds to the use of a store buffer for each shared variable and each process. We also will consider, as an intermediate step, the TSO (Total Store Order) memory model, which corresponds to the use of one store buffer per process.

The proposed approach extends a previously developed verification tool that uses finite automata to symbolically represent the possible contents of the store buffers. Its starting point is a program that is correct for the usual Sequential Consistency (SC) memory model, but that might be incorrect under PSO with respect to safety properties. This program is then first analyzed and corrected for the TSO memory model, and then this TSO-safe program is analyzed and corrected under PSO, producing a PSO-safe program. To obtain a TSO-safe program, only store-load fences (TSO only allows store-load relaxations) are introduced into the program. Finaly, to produce a PSO-safe program, only store-store fences (PSO additionally allows store-store relaxations) are introduced.

An advantage of our technique is that the underlying symbolic verification tool makes a full exploration of program behaviors possible even for cyclic programs, which makes our approach broadly applicable. The method has been tested with an experimental implementation and can effectively handle a series of classical examples.

1 Introduction

Modern multiprocessor architectures optimize accesses to shared memory and, doing so, do not implement the traditional *Sequential Consistency* (*SC*) memory model [1], in which all accesses to the shared memory are immediately visible globally. The exact behavior of these processors with respect to memory accesses is rather complex and is usually only described by a set of typical behaviors in vendor documentation. Nevertheless, formal models that cover the behavior

* This work is supported by the grant 2.4545.11 of the Belgian Fund for Scientific Research (F.R.S.-FNRS).

of many existing processors have been defined. These are usually referred to as *relaxed memory models*, among the most common being *Total Store Order* (*TSO*) and *Partial Store Order* (*PSO*), both defined in [2, 3]. Writing correct code under these models is quite challenging given that they allow even more executions than the traditional SC model. This has motivated work on verifying code under these memory models, as well as on techniques for preserving the correctness of code when it is moved from the SC model to a relaxed memory model. This is done by introducing forced memory synchronizations known as *fences*. However, using fences means forgoing the benefits of the hardware optimizations that lead to relaxed memory models, so the issue is to minimize the number of inserted fences.

In earlier work, [4, 5], we proposed a technique that models TSO using store buffers and uses finite automata to represent the potentially infinite set of possible contents of these buffers. This representation coupled with acceleration techniques similar to those proposed in [6], as well as with the *persistent-set* and *sleep-sets* partial-order reduction techniques [7], allows a full exploration of the state space of programs, including for cyclic programs. In this earlier work both the problem of verifying a program under TSO and of inserting fences to preserve the correctness of a program being moved from SC to TSO are addressed.

This paper focuses on porting a program verified under SC to PSO, while preserving its safety properties. The approach is based on the verification techniques and tool already presented in [4, 5] for TSO. The first contribution of this paper is to extend these techniques and the tool to PSO. One challenge that had to be solved for doing this is that in PSO, a single process writes to several buffers, one for each variable. Thus when dealing with the repetition of cycles, it seems necessary to synchronize the writes to different buffers, hence taking us beyond what can be represented with finite automata. Fortunately, as we will establish in this paper, this synchronization can safely be ignored.

The second contribution of the paper is a method for safely porting programs from SC to PSO. It starts by analyzing and correcting the program under TSO, inserting the necessary memory fences [5]. The fences inserted here are *mfences*, which is what is required to avoid the store-load relaxations possible in TSO. The second step is to move a program safe under TSO to PSO. Here, the approach is similar to the first step, but we only use *sfences*, which are weaker, but sufficient for avoiding the store-store relaxations possible under PSO. We present experimental results that show that the approach is quite effective, can efficiently handle a number of meaningful examples, and compares favorably to other methods proposed for the same problem.

2 Concurrent Programs and Memory Models

We consider a very simple model of concurrent programs in which a fixed set of finite-state processes are interacting through a shared memory. Such a concurrent system is thus defined by a finite set of processes $\mathcal{P} = \{p_1, \ldots, p_n\}$ and a finite set of memory locations $\mathcal{M} = \{m_1, \ldots, m_k\}$, the initial content of the shared memory being defined by a function $\mathcal{I} : \mathcal{M} \rightarrow \mathcal{D}$, $\mathcal{D}$ being the domain of memory values.

The definition of each process p_i includes a finite set of control locations $\mathcal{L}(p_i)$, an initial control location $\ell_0(p_i) \in \mathcal{L}(p_i)$ and a set of transitions $\mathcal{T}$ labeled by operations taken from a set $\mathcal{O}$. The transitions of a process p_i are thus elements of $\mathcal{L}(p_i) \times \mathcal{O} \times \mathcal{L}(p_i)$, also written as $\ell \overset{op}{\to} \ell'$, where both $\ell, \ell' \in \mathcal{L}(p_i)$.

The set $\mathcal{O}$ of operations contains the two following memory operations:

– $store(p, m, v)$, the meaning of which is that process p stores the value $v \in \mathcal{D}$ to the memory location m,
– $load(p, m, v)$, the meaning of which is that process p loads the value stored in memory location m and checks if that value is equal to v. The operation is possible only if the values are equal, otherwise it does not go through and execution is blocked.

Under the SC memory model, the semantics of such a concurrent program is the one in which the possible behaviors are all the interleavings of the operations executed by the different processes, and in which the store operations become immediately visible to all processes.

In TSO, each process executing a store operation can directly load the value saved by this store operation, but other processes cannot always immediately see that value and might read an older value stored in shared memory. This is known as the fact that TSO allows the *store-load* relaxation. PSO also allows such *store-load* relaxations to happen but, additionally, stores accessing different shared memory locations can be reordered within the same process, which is known as the *store-store* relaxation. Thus, the possible SC-executions are included in the set of TSO-executions, which are themselves included in the set of PSO-executions.

The formal definitions of the memory models use the concepts of *program order* and *memory order* [2, 8]. Program order ($<_p$) is a partial order in which the instructions of each process are ordered as executed, but instructions of different processes are not ordered with respect to each other. Memory order ($<_m$) is a total order on the memory operations, which is fictitious but characterizes what happens during relaxed executions.

Let l or l^i denote any load operation, s any store operation, l_a a load operation on location a, and s_a or s_a^i store operations on location a. Furthermore, let $val(l)$ be the value returned by the load operation l.

Using these notions, a formal definition of PSO can be given (for the definitions of SC and TSO, see [2, 8] or [5]).

A PSO execution is one for which there exists a memory order satisfying the following constraints for each process p:

1. $\forall l^1, l^2 : l^1 <_p l^2 \Rightarrow l^1 <_m l^2$
2. $\forall l, s : l <_p s \Rightarrow l <_m s$
3. $\forall s_a^1, s_a^2 : s_a^1 <_p s_a^2 \Rightarrow s_a^1 <_m s_a^2$
4. $val(l_a) = val(\max_{<_m}\{s_a \mid s_a <_m l_a \vee s_a <_p l_a\})$. If there is no such a s_a, $val(l_a)$ is the initial value of the corresponding memory location.

The first three rules specify that the memory order has to be compatible with the program order, except that a store can globally be postponed after a later load or a later store accessing a different variable of the same process. The last rule specifies that the value retrieved by a load is the one of the most recent store in memory order that precedes the load in memory order or in program order, the latter ensuring that a process can see the last value it has stored. If there is no such store, the initial value of that memory location is loaded.

This axiomatic definitions of PSO gives insight, but the equivalent operational model is much more useful for applying explicit state-space exploration techniques. This operational model is described in Fig.1. Stores from each process are buffered, a separate buffer being used by each process for each shared memory location. A store only takes effect when it is transferred from a buffer to the shared memory, which is called a *commit*. This can be seen as the moment when it is entered into the memory order. A commit operation is an internal system operation, which is assumed to be executed nondeterministically for each buffer and each process. This model (using buffers and commits) ensures that stores by the same process accessing the same locations cannot be reordered, while those accessing different locations can. When a load is executed by a process, it will read the most recent value out of its own store buffer for this variable if there exists at least one buffered store to that variable, otherwise the load reads the value out of the shared memory. This means that loads can be reordered with earlier stores of the same process, while they always read the most recent values either from a buffer or the main memory.

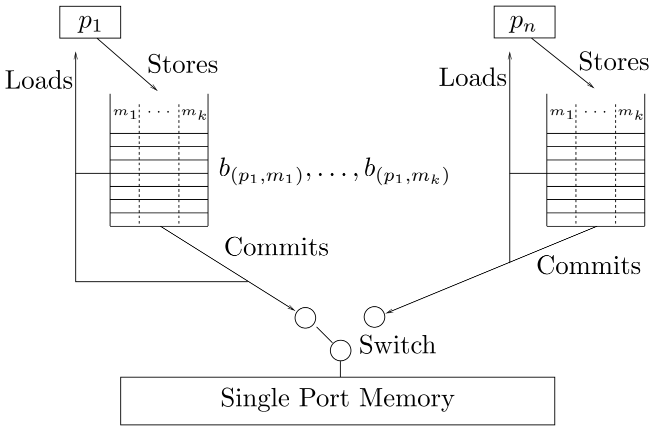


Fig. 1. Operational definition of PSO [2, 3]

To match what is available in actual processors, in particular Intel's x86 processors, extensions have to be made to TSO and PSO [9, 10]. The first extension is adding a new component, the lock, which is used to grant processes exclusive access to the shared memory. The second extension consists of operations called memory fences, which constrain how stores are committed to main memory. In TSO, only one type of fence is available, the *mfence*. An *mfence* operation blocks the executing process until every earlier executed store operation of that process has been committed to the shared memory. In PSO, a second type of fence is also available, the *sfence*. When an *sfence* occurs in a process, it forces every store preceding the *sfence* to be committed to memory before every store that occurs after the *sfence*. An *sfence* does not block the process executing it, but of course restricts the execution of commit operations. When comparing sfences and mfences, it is clear that the effect of an mfence is stronger than the effect of an sfence. The mfence disables all relaxations between operations before and after the mfence, whereas the sfence only disables the *store-store* relaxations.

To formally define the operational model of PSO, we first add a set

$$\mathcal{B} = \{b_{(p_1,m_1)}, \ldots, b_{(p_1,m_k)}, b_{(p_2,m_1)}, \ldots, b_{(p_n,m_k)}\}$$

of buffers to the system, each process having one store buffer per variable[1]. Secondly, we add a global lock L component whose value can be a process $p \in \mathcal{P}$ when p holds the lock, or undefined ($\perp$) when the lock is not held by a process. A global state of the system becomes the composition of the content of the memory, the value of the global lock, and, for each process p, a control location as well as the content of its store buffers $[b_{(p,m_1)}, \ldots, b_{(p,m_k)}]$. The content of a buffer is a sequence of elements that are either (1) triplets (m, v, t) where $m \in \mathcal{M}$, $v \in \mathcal{D}$ and $t \in \mathcal{T}$, representing a store operation and identifying the transition where it was executed, or (2) a special symbol $\star^t$ representing an *sfence(p)* transition t. These semantics are very similar to those that were given for TSO in [5], and thus we will focus only on the operations that are specific to, or different in, PSO: *sfence* and *commit*.

sfence operation : *sfence*(p):

$$\forall m \in \mathcal{M} : [b_{(p,m)}] \leftarrow [b_{(p,m)}] \star^t,$$

where t is the transition corresponding to the current *sfence* operation.

commit operation : *commit*(p, m):

If $([L] \neq \perp$ and $[L] \neq p)$, where L is the lock, then *commit*(p, m) cannot be executed;

otherwise, let $[b_{(p,m)}] = (m, v_1, t_1)(m, v_2, t_2) \ldots (m, v_f, t_f)$ (the first element to commit is not an *sfence*). Then, if $[b_{(p,m)}] \neq \varepsilon$, the result of the commit operation is $[b_{(p,m)}] \leftarrow (m, v_2, t_2) \ldots (m, v_f, t_f)$ and $[m] \leftarrow v_1$, or, if $[b_{(p,m)}] = \varepsilon$, the commit operation has no effect. If $[b_{(p,m)}] = \star^t (m, v_1, t_1) \ldots (m, v_f, t_f)$, i.e. the buffer content starts with the symbol representing the *sfence*(p) operation of transition t, then *commit*(p, m) becomes a synchronized operation which requires all buffers of p to start with $\star^t$. If this is not the case, the commit cannot be executed. If all buffers start with $\star^t$, the commit operation can be executed, and simultaneously removes the element $\star^t$ from all buffers.

[1] Note that we introduce the buffers per *process* rather than by *processor*. This approach is safe for verification since it allows more behaviors than a model in which some processes could share the same buffer. Furthermore, it is impossible to know which process will run on which processor when analyzing a program.

Note that *commit(p,m)* is not an operation that can appear in a program, but is assumed to be always enabled and nondeterministically interleaved with the actual program operations. Thus, when an *mfence(p)*, *unlock(p)* or the *sfence(p)* operation is blocked because the buffers of p are not all empty, or because not all buffers of p start with the same $\star^t$, the implicit execution of *commit(p,m)* operations makes it possible to empty the buffers of p or to reach $\star^t$ for all buffers of p, and enable the operation.

3 Representing Sets of Buffer Contents and State Space Exploration

Verifying a program under the TSO or PSO memory models can be done with a tool such as SPIN ([11]). However, this leads to two problems. First, one must bound the size of the buffers in order to keep the model finite-state. Second, the size of the state space quickly explodes as the size of the buffers grows.

These problems were addressed in [4], for TSO, as follows. To start with, rather than limiting buffers to a fixed size, finite automata, called buffer automata, are used to represent possibly infinite sets of buffer contents. Such buffer automata represent sets of unbounded buffer contents, those contained in the accepted language ($L(A)$) of the buffer automata (A). This allows unbounded buffer contents to be taken into account and, with the help of acceleration techniques similar to those of [12] and [6], to explore the full state space of programs, even if they include memory accesses, in particular memory writes, in cycles that can be infinitely repeated. The cycles that actually need to be, and can be, "accelerated" are those in which one particular process repeatedly writes to memory, thus potentially leading to an unbounded buffer content.

For PSO, the situation is similar, except that we need to handle not just one buffer per process, but a set of buffers, one for each variable and that we also need to handle *sfence operations*. The state-space exploration, including the use of partial-order techniques, as well as the detection of cycles is done exactly as for TSO, see [4]. What changes are the operations applied to the buffer automata to accelerate the cycles: rather than operating on a single automaton for each cycle, the one corresponding to the active process, we need to operate on multiple automata, one for each updated variable of the active process. The obvious way to do this is to filter from the cycle the operations corresponding to each variable and only consider these when dealing with the corresponding buffer automaton. This is straightforward to implement, but generates more buffer contents than can actually occur: the link between the number of times write operations are applied to different variables is lost! To make this clear, let us examine an example.

Consider the program given in Fig 2. It contains just one process with memory locations x, y and z all set to 0 initially. There will be a cycle detected after the sequence of states $1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 1$, and the content of the buffers for x, y and z will then be modified to be $((x,1,t_1)(x,1,t_1)^*; (y,1,t_2)(y,1,t_2)^*; \varepsilon)$. However, since the number of stores to x and y are the same, the accurate representation of the buffer contents after iterating the cycle would be $((x,1,t_1)(x,1,t_1)^n;$

$(y, 1, t_2)(y, 1, t_2)^n; \varepsilon)$, and thus by considering the variables separately we have introduced buffer contents that cannot be generated by iterating the cycle. Fortunately, this is not a problem since committing several times the same memory write operation has no influence on the possible future behaviors of the program. More precisely, any program behavior that is possible from a global state with buffer contents $((x, 1, t_1)(x, 1, t_1)^{n_1}; (y, 1, t_2)(y, 1, t_2)^{n_2}; \varepsilon)$ with $n_1 \neq n_2$ is also possible from the corresponding global state with buffer contents $((x, 1, t_1)(x, 1, t_1)^{\max(n_1,n_2)}; (y, 1, t_2)(y, 1, t_2)^{\max(n_1,n_2)}; \varepsilon)$ by applying different numbers of commit operations to the variables x and y.

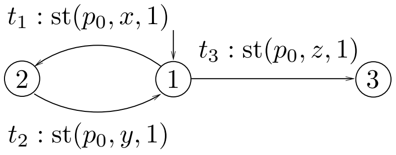


Fig. 2. A program with writes to different variables in a cycle

We now need to generalize the observation made in the previous example. To do this, we have to compare the executions that are possible if we compute the buffer contents resulting from the repeated execution of a cycle separately for each variable, or if we take into account the necessary *synchronization* of the operations performed on the different variables. We will refer to these as *synchronized* versus *unsynchronized* executions. For this we use the following concepts.

Definition 1. *Given a word w over an alphabet Σ and $L \subseteq \Sigma^+$, a word w' is a* ***L* stutter subword** *of w if w can be obtained from w' by, for one or more subwords u of w with $u \in L$, replacing u by a word in u^+.*

Example. The word *aabc* is a $\{b, c, bc\}$ stutter subword of *aabbbcc* and *aabcbcbc*.

Definition 2. *A sequence of operations that does not modify the store buffer in a way that affects the result of subsequent load operations is called* **load-preserving**.

We can then formalize the fact that repeating load-preserving sequences of commit operations has no real impact on an execution.

Lemma 1. *Le σ be an execution of a concurrent system and let LE be the set of load-preserving commit operation sequences appearing in σ. Then every LE stutter subword σ' of σ is also a valid execution of the system.*

Proof. This is a direct consequence of the fact that load-preserving sequences of commit operations are idempotent, i.e. applying them one or several times has no effect on the rest of the execution.

From this Lemma, it is easy to establish the property we need.

Theorem 1. *Computing the buffer automata of different variables independently only leads to valid executions.*

Proof. Indeed, the potentially incorrect executions that could be obtained by handling the buffers for different variables independently are those in which the number of stores to variables executed in the same cycle could be taken to be different. Notice that this will only have an effect on the execution when these stores are committed to memory and that committing the stores appearing on a cycle is load-preserving. Thus, such an unsynchronized execution will always be a LE stutter subword of a synchronized execution, where LE is the set of load-preserving commit sequences corresponding to cycles, and hence will be valid. Indeed, since we allow unbounded repetition of cycles, the synchronized execution can be taken to be the one in which the cycle is repeated a number of times greater than the largest number of times a store to any of the variables modified in the cycle is committed to memory.

After having introduced buffer automata representing sets of buffer contents rather than single buffer contents, one needs to redefine the operations on buffers to also apply to buffer automata. For the operations *store* and *mfence*, please refer to [4, 5].

load operation : *load*(p, m, v):

The problem with a load operation applied to a buffer automaton is that it may succeed on some contents of the buffer represented by the automaton and fail on others. Thus, once a load was successfully applied to a buffer automata, we need to restrict the possible buffer contents to those on which the load operation succeeds (see [4]). But now that we are dealing with PSO, special care should be applied if $\star^t$ symbols are present. Indeed, if a $\star^t$ symbol is removed when modifying a buffer to take into account the fact a load has succeeded, the synchronization required by the *sfence* will no longer be possible, thus introducing a fictitious deadlock. If this occurs the buffers for the other variables of the process are also modified in order to remove the now spurious $\star^t$ symbols.

sfence operation : *sfence*(p):

$$\forall m \in \mathcal{M} : L(A_{(p,m)}) \leftarrow L(A_{(p,m)})\ \star^t,$$

where t is the transition corresponding to the current *sfence* operation.

commit operation : *commit*(p):

As for the load operation, the commit also may have an impact on the possible buffer contents. How to restrict the buffer contents to those that match the current commit operation has been described in [4]. The related problem due to the sfences, as described above for the load operations, also occurs for the commit operation and is handled similarly.

4 From SC to TSO to PSO

We now turn to the problem of preserving the correctness of a program when it is moved from an SC to a PSO memory environment. By correctness, we mean preserving state (un)reachability properties. Note that this captures safety properties, since safety can always be reduced to state (un)reachability in an extended model.

An obvious way to make sure a program can safely be moved from SC to PSO is to force writes to be immediately committed to main memory by inserting an mfence after each store, thus precluding any process from moving with a nonempty store buffer. The obvious drawback of doing so is that any performance advantage linked to the use of store buffers in the implementation is lost.

However, it is not at all necessary to guarantee that the executions that can be seen under PSO are also possible under SC. We might rather just restrict the possible executions to those satisfying the desired safety property, i.e. only exclude those executions reaching states violating the safety property. Recall that the difference between SC, TSO and PSO can be summarized as follows: SC does not allow any relaxation, TSO allows the *store-load* relaxation, and PSO allows the *store-load* and the *store-store* relaxations. When needed, these relaxations can be avoided by placing adequate fences into the program.

In [5], we exploited this to maintain correctness of a program (wrt a safety property) when it was moved from SC to TSO. In the current approach, we want to go further and maintain correctness of a program when it is moved from SC to PSO. We will do this by first modifying the program to guarantee that it is still correct under TSO, and then further modify it so that it remains correct under PSO.

To avoid all relaxations, it is sufficient to place an *mfence* between all loads and any preceding store, as well as an *sfence* between stores accessing different variables. If this is the case, no relaxation will be possible, and all PSO executions will also be SC executions. As our approach proceeds in two steps, the first of which is described in [5], we now only need to describe how to avoid the *store-store* relaxations allowed in PSO, but not in TSO. Lemma 2 gives a sufficient condition for guaranteeing this..

Lemma 2. *Given a PSO execution, if in the program order of each process, an* sfence *is executed between every pair of successive stores accessing different memory locations, the memory order satisfies all the TSO constraints.*

Proof. The semantics of sfence operation can be formalized by introducing these operation in the memory order with the following constraints, where s_a represents a store operation accessing memory location a, and S represents an *sfence* operation:

1. $\forall s_a, S : s_a <_p S \Rightarrow s_a <_m S$
2. $\forall s_a, S : S <_p s_a \Rightarrow S <_m s_a$

In the conditions of the lemma, we have if $s_a <_p s_b$, there is an *sfence* S such that $s_a <_p S$ and $S <_p s_b$, and thus we have that $s_a <_m s_b$. It follows that the memory order thus satisfied all constraints of a TSO order. □

Combining the criteria of lemma 2 with the one of [5], we obtain a sufficient condition for guaranteeing correctness while moving from SC to TSO to PSO. The condition is expressed on executions, but can easily be mapped to a condition on programs: in the control graph of the program, an *mfence* (resp. *sfence*) must be inserted on all paths leading from a store to a load (resp. a store to a store accessing different variables). This is sufficient, but can insert many unnecessary *mfence*/*sfence* instructions. We now turn to an approach that aims at only inserting the fence instructions that are needed to correct errors that have actually appeared when moving the program from SC to TSO to PSO.

5 An Iterative Fence Insertion Algorithm

The basic outline of the algorithm is quite simple: it consists of two steps, and is based on the iterative algorithm of [5]:

1. apply the iterative algorithm of [5] for TSO, starting with a safe program P under SC and returning a TSO-safe program P', by inserting only mfence instructions into the program;
2. apply the iterative algorithm of [5] adapted as described below for PSO, starting with the TSO-safe program P' and returning a PSO-safe program P'', by inserting only sfence instruction into the program.

The algorithm will thus first make the program correct under TSO by iteratively inserting mfence operations. When this is done, the TSO-safe program is analyzed under PSO, and sfence operations are inserted iteratively until the program is correct under PSO. Both parts are guaranteed to terminate, see [5] for the first step and lemma 2 for the second step.

In this second step, the idea is still to look for relaxations (this time we look for *store-store* relaxations) that occur on a path that leads to an error state. To detect *store-store* relaxations, we need to keep track of which operations are compatible with TSO and which are not. This is done by running the state-space exploration with TSO store buffers alongside the PSO store buffers. All operations are also applied to the TSO-buffers, until a *store-store* relaxation is encountered. Once such a relaxation is encountered, we stop updating the TSO-buffer for the process for which the relaxation has occurred since the execution no longer is a TSO-execution, while continuing to update the TSO-buffers for the other processes. Note however that once the TSO-buffer stops being updated for a process, updating can be restarted when all PSO-buffers of that process are completely empty, the TSO-buffer being then reset to empty.

A *store-store* relaxation is detected as follows. The set of enabled transitions of a given global state is computed using the PSO-buffers, which allows the memory order of stores to be changed. When the order of two stores is changed, i.e. a commit of a store is executed while an earlier store accessing another variable is still in the corresponding buffer, the commit of the later store cannot be executed on the TSO-buffer, which indicates that a relaxation has occurred, and the state is marked as a *store-store* relaxation. This relaxation can be disabled by placing an sfence operation before the store operation for which the infringing commit has been executed.

When exploring the state-space under PSO, we know that, if we reach an error state, at least one *store-store* relaxation must have occurred on the path leading to that state. It is then sufficient to disable one of these relaxations to remove that path. When there is a choice of relaxations to disable, we choose the latest on the path leading to the detected error state.

Remark 1. Note that we will not necessarily detect all *store-store* relaxations on a path, as our symbolic buffer content representation makes it impossible to keep the TSO-buffer correctly updated once a relaxation has occurred. New iterations will thus be necessary to find all *store-store* relaxations.

Remark 2. The algorithm we have presented does not guarantee that a program with a minimal number of fences is produced. It could happen that, after the algorithm has iteratively inserted a given number of fences, a fence that was inserted becomes unnecessary due to fences inserted later. One could reiterate on the introduced fences by removing a fence and checking if an error state can be reached. If so, the fence is needed, if not, we can safely remove it. After repeating this procedure until no more fences can be removed we obtain a fence set called "*maximal permissive*"[2], meaning that each fence is needed to ensure the safety property. This does not however imply that the set of inserted fences is globally minimal since the set obtained is dependent on the order in which fences are inserted.

Note however, that no inserted sfence can make an mfence unnecessary. Indeed, sfences will not prevent the *store-load* relaxations that can occur in TSO. The reiteration for removing unnecessary fences should thus be done first after inserting mfences to make the program correct under TSO, and then a second time after the insertion of sfences to adapt the program for PSO.

6 Experimental Results

The fence insertion technique presented in this paper has been implemented within the prototype tool described in [4], extending the tool presented in [5]. The input language for this tool is a simplified and modified version of Promela. It is implemented in Java and uses the BRICS automata-package [14] for handling the automata representing buffer contents.

This prototype has been tested on examples, most of which are mutual exclusion algorithms (part (a) of Tab. 1). For all those algorithms, we could successfully modify the programs to produce a PSO-safe program, by first iteratively inserting mfence operations in order to make the program TSO-safe, followed by iteratively inserting sfence operations to finally obtain a PSO-safe version of

[2] Which was first defined in [13].

the program. For all those programs, no limitation on the size of buffers were enforced, and most of the programs were analyzed when all processes try to enter into the critical section repeatedly. The only program where only a single entry by each process were considered is Lamport's Bakery, where the use of the counter pushes the repeated entry version beyond the scope of our tool. For those programs, the number of iterations is the sum of inserted mfences and sfences, incremented by 2 (each step (for TSO and then PSO) needs an iteration to build the state-space of the corrected program and check that there are no more errors). The column #St contains the number of states in the state-space of the corrected program (all mfences and sfences inserted). All computed fence sets are maximal permissive, except for Szymanski's algorithm and Lamports fast mutex. For both of these algorithms, one could use Remark 2 to obtain a maximal permissive fence set.

Part (b) of Tab. 1 describes the results for programs that were analyzed and did not need to be corrected to stay correct under TSO or PSO. For those programs, execution times only contains one iteration, which explored the state-space under PSO only, without detecting any error state. All those programs were taken from [15], the Increasing Sequence example being limited to 10 instead of 20.

Table 1. Experimental results for several programs with memory fence insertion

Mutual Exclulsion Algorithms			Corrected PSO-safe program				
Program	entry-vers	#Proc	#St	#it	#mfence	#sfence	t
Dekker	repeated	2	381	6	4	0	1.9s
Peterson	repeated	2	219	6	2	2	1.4s
Generalized Peterson	repeated	3	28544	8	3	3	56.7s
Lamport's Bakery	single	2	727	8	4	2	3.2s
Burns	repeated	2	123	4	2	0	1.2s
Szymanski	repeated	2	221	8	6	0	2.2s
Dijkstra	repeated	2	879	4	2	0	3.9s
Lamport's Fast Mutex	repeated	2	5654	10	4	4	11.3s

(a)

Other programs (PSO-safe)			No fences inserted				
Program	limit	#Proc	#St	#it	#mfence	#sfence	t
Alternating bit	-	2	1184	1	0	0	2.3s
Clh queue lock	-	2	3004	1	0	0	2.8s
Increasing Sequence	10	2	59570	1	0	0	140s

(b)

All experimental results were obtained by running our Java-program on a laptop with a 2.7GHz quad-core processor and 8GB RAM, running Ubuntu.

7 Conclusions and Comparison with Other Work

Other work on verification under relaxed memory models includes [16], which proceeds by detecting behaviors that are not allowed by SC but might occur under TSO (or *PSO*). This is done by only exploring SC interleavings of the program, and by using explicit store buffers. The more theoretical work presented in [17] uses results about systems with lossy fifo channels to prove the decidability of reachability under TSO (or PSO) with respect to unbounded store buffers, but the undecidability of repeated reachability. Another approach adopts the axiomatic definition of relaxed memory models and exploit SAT-based bounded model checking [18–20], which of course pushes handling cyclic programs or unbounded buffers beyond their reach. Yet a different approach can be found in [21], which proposes an approach based on SPIN that uses a Promela model with (bounded) explicit queues and an explicit representation of the dependencies on memory accesses that are implied by the relaxed model *RMO* (*Relaxed Memory Order*) [3]. Finally, [22] presents an approach for the verification of programs under relaxed memory models where finite-state programs under SC may turn into infinite-state programs under TSO that proceeds by under-approximation.

With respect to fence insertion algorithms, several other approaches have been proposed. Note that the main originality of our approach is that it is based on a tool that can analyze cyclic programs under TSO/PSO and thus that it can infer fence insertion in this context.

In [23], an over-abstraction technique for potentially infinite store buffers is proposed, combined with the fence insertion algorithm described as "maximal permissive" that was presented in [13]. The abstraction works by representing the buffers as a combination of a finite fifo-buffer that keeps the order of the stores and of an unordered set of stores that is used when the fifo-buffer is full. The fence inference technique works by propagating through the state graph constraints that represent relaxations that could be removed by an mfence or sfence. Once an undesirable state is reached, one can use the associated constraints in order to determine how to make that state unreachable for all incoming paths. However, even if the state-space that is computed is finite in theory, the number of states grows very fast, even for very simple programs, which puts Lamport's fast mutex out of reach of this method, if a first fence is not manually inserted before running the tool. A version of the CLH queue lock could also not be handled, but it is unclear if their version and ours are the same. Also, the increasing sequence example cannot be verified by their approach. For all programs that both our and their approach can handle, and for which no manual fence insertion was done, the computed fences are the same.

Another important piece of work to mention is [15], which exploits the fact that TSO can be simulated by lossy fifo channels. The advantage is that in this setting, state reachability is decidable by a procedure that can be implemented quite efficiently. This approach, combined with a fence insertion algorithm that computes all minimal fence sets, by restricting the places in the program where fence insertion is allowed, makes it very efficient in the case of TSO. It is worth mentioning that their technique for computing the minimal fence sets is compatible with our approach in the case of TSO, as they iteratively construct those sets by looking for relaxations on a path to an error state. However, in the case of TSO, our approach for inserting mfences iteratively is as optimal as the one in [15], and the number of mfences is consistent with their results. It might be confusing that for Dekker's algorithm, we insert 4 instead of 2 mfences, but this is only caused by a different modeling of the same algorithm.

Finally, the simultaneously appearing [24] presents results based on the idea of "TSO-Robustness", i.e. ensuring by fence insertion that, under TSO, only executions which correspond to SC-executions are allowed. It does not consider PSO.

As conclusion, we have successfully extended our previous work on relaxed memory models from TSO to PSO, obtaining experimental results that compare favorably with other results on this topic. It came as a pleasant surprise while developing these results that the synchronized writing to different buffers that at first seemed necessary and impossible to handle simply, was in fact not needed.

References

1. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9), 690–691 (1979)
2. SPARC International, Inc., CORPORATE: The SPARC architecture manual: version 8. Prentice-Hall, Inc., Upper Saddle River (1992)
3. SPARC International, Inc., CORPORATE: The SPARC architecture manual (version 9). Prentice-Hall, Inc., Upper Saddle River (1994)
4. Linden, A., Wolper, P.: An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
5. Linden, A., Wolper, P.: A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 144–160. Springer, Heidelberg (2011)
6. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The Power of QDDs (Extended Abstract). In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 172–186. Springer, Heidelberg (1997)
7. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
8. Loewenstein, P., Chaudhry, S., Cypher, R., Manovit, C.: Multiprocessor memory model verification. Technical report Unpublished presentation at FLOC-AFM 2006 (2006), http://fm.csl.sri.com/AFM06/papers/4-Loewenstein.pdf
9. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53, 89–97 (2010)
10. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)
11. Holzmann, G.: Spin model checker, the: primer and reference manual, 1st edn. Addison-Wesley Professional (2003)
12. Boigelot, B., Wolper, P.: Symbolic Verification with Periodic Sets. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 55–67. Springer, Heidelberg (1994)

13. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Austin, TX, pp. 111–120. FMCAD Inc. (2010)
14. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010), http://www.brics.dk/automaton/
15. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-Example Guided Fence Insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
16. Burnim, J., Sen, K., Stergiou, C.: Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 11–25. Springer, Heidelberg (2011)
17. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 7–18. ACM, New York (2010)
18. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 12–21. ACM, New York (2007)
19. Burckhardt, S., Musuvathi, M.: Effective Program Verification for Relaxed Memory Models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
20. Burckhardt, S., Alur, R., Martin, M.M.K.: Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 489–502. Springer, Heidelberg (2006)
21. Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). SIGARCH Comput. Archit. News 36, 65–71 (2009)
22. Atig, M.F., Bouajjani, A., Parlato, G.: Getting Rid of Store-Buffers in TSO Analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
23. Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 187–198. ACM, New York (2011)
24. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and Enforcing Robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)

Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory

David H. White and Gerald Lüttgen

Software Technologies Group, University of Bamberg, Germany
{david.white,gerald.luettgen}@swt-bamberg.de

Abstract. We investigate whether dynamic data structures in pointer programs can be identified by analysing program executions only. This paper describes a first step towards solving this problem by applying machine learning and pattern recognition techniques to analyse executions of C programs. By searching for repeating temporal patterns in memory caused by multiple invocations of data-structure operations, we are able to first locate and then identify these operations. Applying a prototypic tool implementing our approach to pointer programs that employ, e.g., lists, queues and stacks, we show that the identified operations can accurately determine the data structures used.

Keywords: Program comprehension, pointer programs, dynamic data structures, machine learning, pattern recognition.

1 Introduction

Programs making heavy use of pointers are notoriously difficult to analyse. To do so one needs to understand which dynamic data structures and associated operations the program employs. Analysis tools for pointer programs, such as those based on shape analysis [17] and pointer graph abstraction [8], rely on an abstraction methodology that must be crafted for each specific data structure, and thus require *a priori* knowledge of the program to be analysed.

Knowing the shape of a data structure is, however, sometimes insufficient for understanding its behaviour. For example, to recognise a linked list implementing a stack, the operations that manipulate the data structure are of key importance. Static analyses typically provide only approximations for this type of behaviour, due to imprecision in the analysis. The task is further complicated when dealing with legacy code, programs with unavailable source code and, even worse, programs with obfuscated semantics such as malware. Hence, the question arises whether pointer programs can be understood, with high confidence via a dynamic analysis that identifies dynamic data structures and the operations that manipulate them from an execution trace of the program under analysis.

Identifying (or in machine-learning terminology: *labelling*) operations appearing in a program trace is a difficult problem. The initial obstacle is simply locating data structure operations, i.e., determining which *events* (e.g., a pointer write) in the trace correspond to an operation and which do not. This problem is compounded by the fact that invocations of the same operation may look very different: clearly the addresses appearing in pointer variables will differ, but there may also be significant differences in the control path taken due to traversal or corner cases, such as inserting to an empty data structure.

The key idea is to locate an operation by learning the repetition in the program trace caused by multiple invocations of that operation. For this to work, we must construct an abstraction of the trace that lessens the differences between invocations, and thus exposes the repetition. However, we need to make some realistic assumptions for this to be feasible. Firstly, there should be a sufficiently large number of invocations to expose the repetition, and secondly, the surrounding context of the invocations should vary; otherwise, the context could be included in the repeating pattern.

However, merely locating repetition in the program trace is insufficient as it is highly likely that repetition resulting from non-operations will also be discovered. Furthermore, as there are many different ways to code a data structure operation, it is unlikely that it will be possible to assign a label at the granularity of repeating pattern elements. To solve both problems, we consider an instance of a repeating pattern a *potential operation*. We then construct a snapshot of the pre- and post-memory states of the potential operation, and assign a label based on the difference between these. With the set of data structure operations to hand, identifying the program's data structures is an easy task.

Contribution and Approach. Our contribution is the automated identification of dynamic data structures appearing in an execution trace of a C program via a labelling of the operations that manipulate them. We have written a prototypic software tool to evaluate our approach on a number of textbook programs implementing dynamic data structures, in addition to real-world examples. The current prototype employs user-specified templates to identify iterative data structures such as lists, queues, stacks, etc., and has a couple of limitations that should be addressed by future work: nested data structures/operations are not handled and patterns for non-tail recursive operations cannot be learned.

We divide the description of our approach into three parts: Sec. 2 shows how we compute a suitable abstraction from an execution, Sec. 3 introduces the machine learning of repetition, and Sec. 4 explains the labelling process for operations and data structures. We begin Sec. 2 by describing the type of events we wish to capture from an execution, in addition to how the instrumentation is performed. Thus, the execution of the instrumented source code gives an *event trace*. For each event we compute a *points-to graph*, and the sequence of these is the *points-to trace*. However, the points-to trace is unsuitable as input to the machine learning as the specific information about an event is captured very inefficiently. Therefore, we construct a second abstraction for each points-to graph that captures the semantics of the event; using machine learning terminology we term this abstraction a *feature*.

The search for repeating structure takes place on the *feature trace*, where the goal is to learn the set of *patterns* that best captures the repetition (Sec. 3). The notion of "best" is determined by a *Minimum Description Length* [6] criterion that evaluates how successful a set of patterns is at compressing the feature trace. The search is performed using a genetic algorithm, which is particularly good at finding globally good solutions. Each *occurrence* of a pattern corresponds to a potential operation, and labelling is performed by matching against a repository of *templates* for known data structure operations (Sec. 4). Data structure labelling is then achieved by considering the set of operations that manipulated a connected component in the points-to graph.

Our prototype tool is implemented using a combination of C++, the C Intermediate Language (CIL) [12] and Evolving Objects [4] (11k LOC) and took nine person-months to develop. It is employed to evaluate the effectiveness of our approach at locating and labelling data structure operations written in C (Sec. 5). We first consider data structure source code taken from textbooks [3, 18–20]. We then apply the tool to real-world programs [1, 11, 13]. Finally, in Sec. 6, we discuss related work, give conclusions and describe future work.

2 Trace Generation and Preprocessing

In this section we present the construction of the event trace, points-to trace and feature trace required for our machine learning approach.

Generating the Event Trace. We consider a dynamic data structure to be a set of *objects* (C `struct`s) linked by pointers. There are three types of program events that must be captured in the trace: pointer writes, dynamic memory events, and stack pointer variables leaving scope. To record these events during a program's execution, we instrument the source code using the CIL API [12]. We now describe and motivate each event type.

The abstraction must capture the topology of a data structure, and since the topology is defined by pointer writes and their types, this information must be captured in the trace. All pointer-write events have the following attributes: sourceAddr, targetAddr and pointerType. We differentiate between two types of pointer writes: those occurring in the *context* of an encapsulating object, i.e., assignments to `context.ptr` or `context->ptr`, where the `context` is the `struct` in which the pointer appears, and those with no context. If the write has context, then the predicate *hasContext* on the event is true and two additional attributes are set, namely encapsulatingObjectAddr and encapsulatingObjectType.

The deallocation of memory is also key to the abstraction. After a memory region has been deallocated, any information the abstraction was tracking about this region should be disregarded. Attributes for this event type record the beginning and end of the memory region: bFreeAddr and eFreeAddr, respectively. Memory allocations are not recorded as separate events and are instead combined with the pointer write storing the allocation's return value. For writes of this type, the predicate *isAlloc* is true and the attribute allocSize is defined.

We want to understand how the operations affect the data structures beyond the internal modifications; consider removing the front element of a linked list by only updating the head pointer. To identify such modifications we must track the *entry points* to the dynamic data structure. This is simple as we already

```
1  typedef struct node *N_ptr;
2  typedef struct node {
3     int key;
4     N_ptr next, prev;
5  } Node;
6
7  void dllInsertFront(
8     N_ptr *list, int key) {
9     N_ptr new
10       = malloc(sizeof(Node));
11    new->key = key;
12    new->next = *list;
13    new->prev = NULL;
14    if (*list != NULL)
15       (*list)->prev = new;
16    *list = new; }
```

Fig. 1. An operation to insert in the front of a doubly linked list

record all pointer writes; however, care must be taken if the pointer write is in the stack as this memory has a lifetime defined by its scope. Thus, events of this type store the address of the pointer variable leaving scope in attribute varAddr.

To exemplify our approach, we give a running example based on the insert-front doubly linked list operation in Fig. 1. It executes in one of two *modes*, inserting to the front of an empty list or a non-empty list. Instrumentation will be inserted at lines 9, 12, 13, 15 and 16 to record pointer writes, and after line 16 to record local variables that go out of scope.

Constructing the Points-to Trace. For each event in the event trace $\langle E_1, \ldots, E_n \rangle$, a points-to graph is constructed that describes the effect of that event on the memory state. Points-to graph G_i is constructed by applying event E_i to points-to graph G_{i-1}, where the initial points-to graph is G_0.

A points-to graph $G = (\mathcal{V}, \mathcal{E})$ is composed of a vertex set $\mathcal{V}$ and an edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. There is exactly one of each of the following three special vertices in each points-to graph: v_{null}, the target of null pointers; v_{undef}, the target of undefined pointers; and $v_{\text{disconnect}}$, a vertex with no edges that is used as a placeholder return value. All remaining vertices represent objects, and each has the following set of attributes obtained from the event trace: a beginning address (bAddr), an end address (eAddr) and a type (type). A type t has a set of pointer fields $\{f_1, f_2, \ldots\} = t.\text{fields}$. A compound variable object may have any number of pointer fields (including zero), while a raw pointer has exactly one; raw pointers are used as entry points to the data structure. Each field has an associated type ($f_i.\text{type}$) and offset ($f_i.\text{offset}$). An edge $e \in \mathcal{E}$ represents a pointer and has a source address attribute (sAddr). We do not require that the source addresses of the out-edges of $v \in \mathcal{V}$ be compatible with the field offsets given by $v.\text{type}$.

The pseudocode in Fig. 2 describes how the points-to graph is updated for an event. A pointer write provides two opportunities for adding information to the points-to graph beyond adding the written pointer. If the write has context, then we can add information about the object encapsulating the pointer, and we may always add information about the target object based on the pointer type. This occurs between lines 2-10 of Fig. 2 and in findOrAddVertex. Now, the vertices representing the source and target objects of the pointer are stored in v_s

```
 1: if $isPointerWrite(E)$ then
 2:   if $hasContext(E)$ then
 3:     $A \leftarrow E.$encapsulatingObjectAddr; $T \leftarrow E.$encapsulatingObjectType
 4:   else
 5:     $A \leftarrow E.$sourceAddr; $T \leftarrow E.$pointerType
 6:   $v_s \leftarrow \textsc{findOrAddVertex}(A, T)$
 7:   if $E.\text{targetAddr} \neq$ NULL then
 8:     $v_t \leftarrow \textsc{findOrAddVertex}(E.\text{targetAddr}, \textsc{deref}(E.\text{pointerType}))$
 9:   else
10:     $v_t \leftarrow v_{\text{null}}$
11:   $\mathcal{E} \leftarrow \mathcal{E} - \{e \in \mathcal{E} : e.\text{sAddr} = E.\text{sourceAddr}\} \cup \{(v_s, v_t)\langle E.\text{sourceAddr}\rangle\}$
12: else
13:   if $isMemoryFree(E)$ then
14:     $\mathcal{V}_{\text{remove}} \leftarrow \{v \in \mathcal{V} : [v.\text{bAddr}, v.\text{eAddr}) \subseteq [E.\text{bFreeAddr}, E.\text{eFreeAddr})\}$
15:   else if $isVarOutOfScope(E)$ then
16:     $\mathcal{V}_{\text{remove}} \leftarrow \{v \in \mathcal{V} : v.\text{bAddr} = E.\text{varAddr}\}$
17:   for all $v \in \mathcal{V}_{\text{remove}}$ do
18:     $\mathcal{E} \leftarrow \mathcal{E} - \textsc{edges}(v) \cup \{(v_s, v_{\text{undef}})\langle (v_s, v_t).\text{sAddr}\rangle : (v_s, v_t) \in \textsc{inEdges}(v)\}$
19:   $\mathcal{V} \leftarrow \mathcal{V} - \mathcal{V}_{\text{remove}}$

20: procedure $\textsc{findOrAddVertex}(A : Address, T : Type)$
21: if $\exists v \in \mathcal{V} : [A, A + T.\text{size}) \subseteq [v.\text{bAddr}, v.\text{eAddr}])$ then return $v$
22: else
23:   $v_{\text{new}} \leftarrow \textsc{createVertex}(\text{type} = T, \text{bAddr} = A, \text{eAddr} = A + T.\text{size})$
24:   for all $v \in \mathcal{V} - \{v_{\text{new}}\} : [v.\text{bAddr}, v.\text{eAddr}) \subseteq [v_{\text{new}}.\text{bAddr}, v_{\text{new}}.\text{eAddr})$ do
25:     $\mathcal{E} \leftarrow \mathcal{E} - \textsc{inEdges}(v) \cup \{(v_s, v_{\text{new}})\langle (v_s, v_t).\text{sAddr}\rangle : (v_s, v_t) \in \textsc{inEdges}(v)\}$
26:     $\mathcal{E} \leftarrow \mathcal{E} - \textsc{outEdges}(v) \cup \{(v_{\text{new}}, v_t)\langle (v_s, v_t).\text{sAddr}\rangle : (v_s, v_t) \in \textsc{outEdges}(v)\}$
27:     $\mathcal{V} \leftarrow \mathcal{V} - \{v\}$
28:   forall $f \in T.\text{fields}$ do
29:     if $\nexists e \in \textsc{outEdges}(v_{\text{new}}) : e.\text{sAddr} = A + f.\text{offset}$ then
30:       $\mathcal{E} \leftarrow \mathcal{E} \cup \{(v_{\text{new}}, v_{\text{undef}})\langle A + f.\text{offset}\rangle\}$
31:   return $v_{new}$
```

Fig. 2. Updating of the points-to graph based on event E

and v_t, respectively. Using this, the edge representing the pointer is added and any pre-existing edge for this pointer is removed (line 11). We use the notation $e\langle \text{A} \rangle$ to initialize the source address attribute of edge e to address A.

$\textsc{findOrAddVertex}(A, T)$ returns the vertex that represents the memory needed by type T starting at address A. If there is no suitable pre-existing vertex, then one is added (line 23). However, there may be pre-existing vertices representing subsections of the region, and any information stored by these vertices must be aggregated into the vertex of the new larger region. This process is performed in lines 24-27 where, for each defunct vertex, in-edges are updated to point to the new vertex, out-edges are added to the new vertex, and lastly, the vertex is removed. Finally, for any field of the new vertex's type that does not already have a pointer, an edge is added from that field to v_{undef} (lines 28-30).

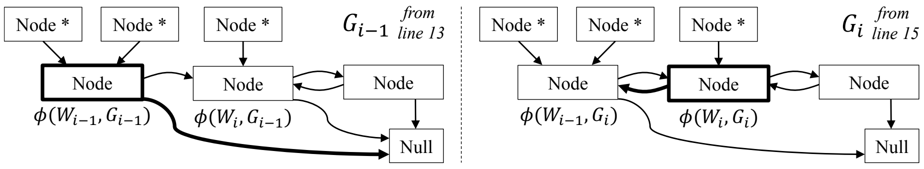


Fig. 3. Points-to graphs generated from the code in Fig. 1. The highlighted pointer is written in the event, and the highlighted vertex is the written vertex (discussed later). The vertices labelled "Node *" are entry points to the data structure.

Deallocation and variable-out-of-scope events are handled in lines 13-19. The only distinction is that a deallocation event may remove a set of vertices, while an out-of-scope event will remove only one vertex. If there were any in-edges to a removed vertex, then the edges' targets are set to v_{undef} (not shown in Fig. 3).

Fig. 3 displays the points-to graphs after the pointer writes on line 13 and 15 of Fig. 1 have been performed on the third call to `dllInsertFront()`.

Constructing the Feature Trace. We construct a *feature trace* $F = \langle F_1, \ldots, F_n \rangle$, where F_i captures the effect of E_i in a way that exposes repetition in the trace. A feature F_i is composed of two types of sub-features: structural sub-features that abstractly describe the change in local topology between G_{i-1} and G_i, and temporal sub-features that capture the relationship between the written pointers in E_{i-1} and E_i.

Let $W_i = E_i.\text{sourceAddr}$ if $\mathit{isPointerWrite}(E_i)$; otherwise, W_i is set to a dummy value that will never be used as an address. We term the vertex in graph G_i that contains address W_i the *written vertex*. In general, a vertex v in a graph G containing address A is computed as follows: $\phi(A, G) = v$ if $\exists v \in \mathcal{V} : A \in [v.bAddr, v.eAddr)$, otherwise $\phi(A, G) = v_{\text{disconnect}}$ (note that there is at most one vertex representing a particular address).

Each structural sub-feature records one aspect of the incoming or outgoing edges of the written vertex. Some sub-features concern the pointer arrangement before the event was performed (i.e., before E_i and calculated on G_{i-1}), and some the arrangement afterwards (i.e., after E_i and calculated on G_i). Further discrimination of pointers is based on the type of the two objects they connect, and whether the pointer is null or undefined. If the event is a memory allocation or free, then additional features are calculated. The full list of features can be seen in Table 1, including example values for the event shown in Fig. 3. The sub-features for out-pointers constructed on the post-state of E_i deserve discussion. Here, additional discrimination is performed based on whether the source and target objects of a pointer have been in the same connected component of the graph (given by COMP) before the event is performed. The rationale behind these sub-features is to capture components being joined or separated.

The first temporal sub-feature records whether the addresses of the written vertices in E_{i-1} and E_i are the same. Sequences of events where this property is true usually represent traversal. The next sub-feature records whether the written vertex in E_i is reachable from E_{i-1} by following one pointer forwards or

Table 1. This table describes how the feature vector F_i is computed for event E_i. To save space some rows represent multiple features ($\bowtie \in \{=, \neq\}$). The features are based on properties concerning the written vertex of events E_{i-1} and E_i. We use the following shorthand for the written vertices: $v^j_{\mathrm{pre}} = \phi(W_j, G_{i-1})$ and $v^j_{\mathrm{post}} = \phi(W_j, G_i)$. The right column shows the value of each feature for event E_i depicted in Fig. 1.

Dynamic Memory Features		*Example*
Allocate	**if** $\mathit{isAlloc}(E_i)$ **then** E_i.allocSize **else** 0	0
Deallocate	**if** $\mathit{isFree}(E_i)$ **then** E_i.freeSize **else** 0	0
Pre and Post Event Structural Features, where $x \in \{pre, post\}$		($\bowtie$)
In Pointers	$\lvert\{e \in \textsc{inEdges}(v^i_x) : \textsc{source}(e).\mathrm{type} \bowtie v^i_x.\mathrm{type}\}\rvert$	*pre*: 2(=), 1($\neq$) *post*: 2(=), 1($\neq$)
Null Pointers	$\lvert\{e \in \textsc{inEdges}(v^i_x) : \textsc{target}(e) = v_{\mathrm{null}}\}\rvert$	*pre*: 1, *post*: 0
Undef Point.	$\lvert\{e \in \textsc{inEdges}(v^i_x) : \textsc{target}(e) = v_{\mathrm{undef}}\}\rvert$	*pre*: 0, *post*: 0
Pre-Event Structural Features		($\bowtie$)
Out Pointers	$\lvert\{e \in \textsc{outEdges}(v^i_{\mathrm{pre}}) : \textsc{target}(e).\mathrm{type} \bowtie v^i_{\mathrm{pre}}.\mathrm{type}\}\rvert$	$1(=), 0(\neq)$
Post-Event Structural Features		($\bowtie_1, \bowtie_2$)
Out Pointers	$\lvert\{e \in \textsc{outEdges}(v^i_{\mathrm{post}}) : \textsc{target}(e).\mathrm{type} \bowtie_1 v^i_{\mathrm{post}}.\mathrm{type}$ $\wedge\ \textsc{comp}(v^i_{\mathrm{post}}) \bowtie_2 \textsc{comp}(\phi(\textsc{target}(e).\mathrm{sAddr}), G_{i-1})\}\rvert$	$2(=,=), 0(=,\neq)$ $0(\neq,=), 0(\neq,\neq)$
Temporal Features		
Same Object	$v^{i-1}_{\mathrm{pre}}.\mathrm{sAddr} = v^i_{\mathrm{post}}.\mathrm{sAddr}$	false
Temporal Features, where $x \in \{pre, post\}$		*pre*, *post*
1 Forward	$\lvert\textsc{outEdges}(v^{i-1}_x) \cap \textsc{inEdges}(v^i_x)\rvert > 0$	true, true
1 Backward	$\lvert\textsc{outEdges}(v^i_x) \cap \textsc{inEdges}(v^{i-1}_x)\rvert > 0$	false, true
Component	$\textsc{comp}(v^i_x) = \textsc{comp}(v^{i-1}_x)$	true, true

backwards. The last sub-feature records whether the written vertices in E_i and E_{i-1} are in the same component.

The features described above are sufficiently selective to be able to recognise operations on linked lists and trees (cf. Sec. 5). They are also compact enough to enable an efficient machine learning of patterns. In the following, the exact value vector of a feature will be unimportant; thus, we simply denote features by symbols f_a, f_b, etc., where different indices mean that the features differ.

3 Locating Data Structure Operations in the Trace

We describe repetition in the feature trace in terms of a pattern set, where a pattern is sequentially composed of (i) feature sequences and/or (ii) repetitions of feature sequences. This two-level structure allows repetition to be learned. For example, a feature sequence $[f_a, f_b, f_c, f_d, f_c, f_d, f_e]$ might be represented by the pattern $[[f_a, f_b], [f_c, f_d]^+, [f_e]]$, where the middle sequence is allowed to match multiple times. A feature from a pattern and a feature from the trace match if all sub-features have identical values. There is a small caveat due to the temporal features; we do not require a match of any temporal sub-feature for the first feature in a pattern; this is because we do not want to restrict the matching of a pattern based on the preceding context. As discussed earlier, `dllInsertFront` from Fig. 1 operates in two different modes. Therefore, we would expect these two modes to manifest themselves as two different feature sequences. This is indeed the case as we obtain the two sequences $[f_a, f_b, f_c, f_d]$ and $[f_a, f_e, f_f, f_g, f_h]$. Note that the first feature is identical as the object storing the `malloc` result is disconnected from everything else. The remainder of the sequence diverges due to the differing number of in-/out-pointers to/from the written vertex.

We solve the problem of locating repetition in the feature trace by considering how it may be compressed, the intuition being that the best compression has identified the most repetition. The *Minimum Description Length* [6] (MDL) principle makes this definition precise; it states that the following should be minimized: $L(H) + L(D|H)$, i.e., the length of the hypothesis (the set of patterns chosen to represent the data) summed with the length of the data encoded with the hypothesis. This is a commonly used criteria since it avoids two of the most common pitfalls in machine learning: over-fitting (penalized by the $L(H)$ term) and over-generalizing (penalized by the $L(D|H)$ term).

The MDL criterion determines the *fitness* of a pattern set. We choose a genetic algorithm to explore the space of possible pattern sets as we expect the fitness function to be highly non-continuous. The algorithm proceeds by evolving an initial set of individuals via two operators, *mutation* and *crossover*, until a stopping condition is met. In each generation of the evolution, the fitness of an individual is assessed, and this determines its inclusion in the next generation.

We use a random initialization of the population where each individual is a set of randomly selected patterns. When crossover is applied to a pair of individuals (with probability GA_c), some of the patterns from each are swapped to the other. When mutation is applied to an individual (probability GA_m), a random pattern is selected and one of two operations is applied: (i) the front or back of the pattern is extended or contracted; or (ii) if the pattern contains consecutively repeating subsequences, then these are collapsed into a single instance that is allowed to match multiple times. The front or back of the pattern may only be extended to a feature sequence that occurs in the feature trace. The search terminates when there has been no improvement in the fitness for GA_t generations. Parameters GA_c, GA_m and GA_t are chosen by us as documented in Sec. 5.

4 Labelling Operations and Data Structures

We now determine which potential operations are real data structure operations and then label them, and which are just noise in the trace. The greedy application of the best set of patterns to the feature trace gives the set of potential operations $\mathcal{P}$. An operation $P \in \mathcal{P}$ is a subsequence of the event trace, i.e., $P = \langle E_i, \ldots, E_j \rangle$. Given this definition, $pre(P) = G_{i-1}$ is the points-to graph before the operation was performed and $post(P) = G_j$ is the points-to graph afterwards.

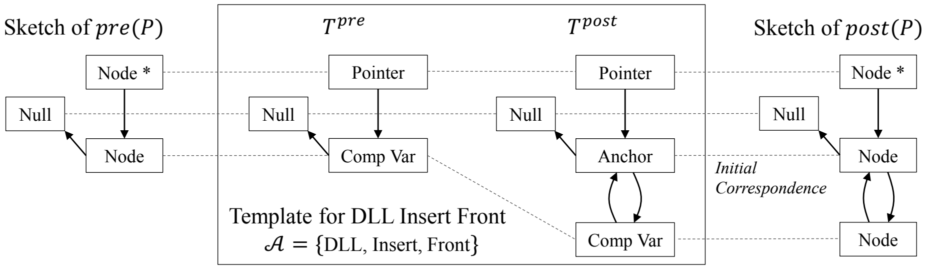


Fig. 4. An example of matching the template for inserting at the front of a doubly linked list. Note that, in T^{pre}, the second pointer to `NULL` is omitted to allow the template to match a DLL insert front operation on a list of any length.

Labelling Operations. We label operations via a template matching scheme, i.e., we manually define a repository of templates $\mathcal{T}$ for the pre and post points-to graphs of any operation we wish to identify, and attempt to match each template in turn. Templates are defined for operations on a singly linked list (SLL), a queue as SLL, a stack as SLL, a doubly linked list (DLL) and a binary tree. There is typically not a 1-1 correspondence between a template match and a real-world data structure operation; so instead of labelling a potential operation directly, a successful match adds a set of attributes to the operation (see below). After a match of all templates has been attempted, the operation label is determined from the identified set of attributes.

Template. A template $(T^{pre}, T^{post}, \mathcal{A}) \in \mathcal{T}$ consists of a pair of template graphs, which are matched against an operation P, and a set of attributes $\mathcal{A}$. A template graph places constraints on which types of template vertices may match which types of points-to graph vertices; we distinguish four types: compound variable vertices, raw pointer vertices (which can be distinguished from compound variable types with one field on the basis of the C types), v_{null} and v_{undef}. One of the template graphs must have more vertices than the other, and the vertex set of the smaller graph is a subset of the larger. Compound variable vertices appearing only in the larger graph are termed *anchor vertices*; at least one of these is always required. An example of a template is given in Fig. 4.

Matching. Anchor vertices are used to provide the initial correspondence(s) for the match. If $|pre(P)| > |post(P)|$ ($|\cdot|$ only counts non-pointer object vertices), then templates with anchor vertices in T^{pre} are applicable for matching; or, if $|post(P)| > |pre(P)|$, then templates with anchor vertices in T^{post} are applicable. Those difference vertices between $pre(P)$ and $post(P)$ that are compound variable vertices provide the set of vertices to be initially mapped to the anchor vertices. With the initial correspondences established, all remaining vertices and edges in the template graph are matched to those in the points-to graph. If this succeeds, and the other template graph can be matched to the other points-to graph given the previous correspondences, then the template is matched. In case there are multiple possible initial mappings to the anchor vertices, all possible permutations are tried. Thus, it does not matter if some of the difference vertices are irrelevant to the operation. Note that our reliance on difference vertices is not a restriction in practice, since all dynamic data structures have some operations (e.g., insert and remove) that exhibit this characteristic and do not have only, e.g., traversing operations. In Fig. 4, $|post(P)| > |pre(P)|$, and T^{post} has an anchor vertex, so T^{post} is first matched to $post(P)$ using the difference vertex for the initial correspondence. Since this matches, we check whether T^{pre} matches $pre(P)$ given the correspondences from the first step. This also matches, and hence, so does the whole template.

Labelling. After all templates have been tested, we examine the set of present and absent attributes now associated with a potential operation to determine its label. Attributes may record *data structures* (SLL, DLL, bTree), *coding style* (Payload, Null-terminated, Header-node, Sentinel-node, Tail-pointer), *mode* (Insert, Remove) and *position* (Front, Middle, End). A formula over the attribute set allows the potential operation to be labelled, e.g., an operation satisfying SLL $\wedge$ ¬DLL $\wedge$ ¬bTree $\wedge$ insert $\wedge$ ¬remove $\wedge$ front $\wedge$ ¬middle is labelled SLL Insert Front, and one satisfying DLL $\wedge$ ¬bTree $\wedge$ insert $\wedge$ ¬remove $\wedge$ front $\wedge$ ¬middle is labelled DLL Insert Front. If the set of attributes is not consistent with exactly one predicate, then we report that operation to the user.

Note that the templates for SLL are easy to match on many non-SLL operations. It is only through a combination of templates and attributes that we are able to handle them correctly. Continuing with our previous example (Fig. 4), a template for SLL Insert Front will also match P, but this is ruled out as the final attribute set will only be consistent with the predicate for DLL Insert Front.

Labelling Data Structures. The final part of this phase is to label the data structures that are manipulated by the operations. For each connected component in the graphs, over the whole points-to trace, we check what combination of operations manipulated this component. This is almost always possible since the set of components is typically stable in real-world programs. If the combination of operations is consistent (within a tolerance t_{ops}, see Sec. 5) for a data structure, then this component is labelled. However, if the component has a severely inconsistent set of operations, e.g., equal numbers of Tree Insert and SLL Insert Middle, then this could be an indication of a programming error. Working out correct combinations is non-trivial since, in special circumstances, one data structure can look like another. For example, if there are many operations for both SLL Insert Front and SLL Insert Front with Header, then the label SLL No Header would be preferred.

5 Evaluation

We first evaluate our prototype tool on data structure source code taken from textbooks [3, 18–20] (SLLs, DLLs, queues, stacks, binary trees). These show that our approach recognizes a number of different data structures and works when operations are coded in different styles. We reinforce these conclusions with experiments on correctly mutated SLL and DLL operations (i.e., permuting statements within the operation in a way that does not change the operation's semantics, e.g., by changing the placing of a call to malloc). Next, we demonstrate that our approach can handle program traces that record events for multiple data structures. Lastly, we apply our approach to real-world programs [1, 11, 13]. All experiments were run under Ubuntu on a modern 16 core PC. The most time consuming step is locating repetition with the genetic algorithm (GA), and this is trivially parallelizable. The largest trace analysed has 15k events and takes 25 minutes (80% spent in GA) and 1GB RAM. GA parameters in Evolving Objects [4] are as follows: $GA_c = 0.1$, $GA_m = 0.1$ and $GA_t = 500$.

Methodology. To evaluate the data structures taken from textbooks, we construct a program for each example to simulate its use in a typical setting. Each program repeatedly chooses an operation applicable to the current state of the data structure to perform. To provide meaningful results, we average the measurements taken over 10 different runs, i.e., simulating the data structure being used in 10 deterministic programs. To make each textbook program more realistic, a randomly chosen "noise" function is sometimes invoked to simulate the program performing other tasks, such as preparing the payload for the data structure. This noise is generated via a set of functions that are indicative of those found in real programs. The noise comprises 30% to 50% of each trace.

When analysing the trace produced by a run, we let $\mathcal{R}$ stand for the set of *real operations*. The set of potential operations that correctly represent a real operation is given by $\mathcal{P}_\mathcal{R} \subseteq \mathcal{P}$. A potential operation has an associated label that is either "NoLabel" or a data structure operation label. Thus, the set of labelled operations is $\mathcal{L} = \{P \in \mathcal{P} : \mathit{label}(P) \neq \text{NoLabel}\}$, and we denote the set of correctly labelled operations by $\mathcal{L}_\mathcal{R} \subseteq \mathcal{L}$.

To evaluate the success of locating repeating patterns, we must compute the set $\mathcal{P}_\mathcal{R}$. This is tricky since potential operations do not need to perfectly map to real operations for the approach to be successful; we consider an overlap of 50% sufficient. Formally, we record that $P \in \mathcal{P}$ is a member of $\mathcal{P}_\mathcal{R}$ if the number of events P has in common (operator $\cap$) with the most appropriate real operation (given by $\psi(P) = \text{argmax}_{R' \in \mathcal{R}}\{|P \cap R'|\}$) is above 0.5. This also enables a definition for the set of *correctly labelled* operations.

$$\mathcal{P}_\mathcal{R} = \{P \in \mathcal{P} : \frac{|P \cap \psi(P)|}{|\psi(P)|} > 0.5\} \quad \mathcal{L}_\mathcal{R} = \{P \in \mathcal{P}_\mathcal{R} : \mathit{label}(P) = \mathit{label}(\psi(P))\}$$

This measure does not penalize potential operations for over-matching a real operation. However, this is only of concern if the user is analysing a program with the labelled operations, and irrelevant parts of the program are included, e.g., if an operation includes noise or part of another operation. We therefore introduce a second measure for the usefulness of a labelled operation to the user, where each summand expresses the proportion of the operation that agrees with the most appropriate real operation, minus the proportion that disagrees:

Table 2. Results from applying our tool to several programs. Programs marked with (without) † have results averaged over 5 (10) runs. Symbol $*$ denotes a program that is currently at the limits of our approach. Superscript X^H means data structure X uses a header node. For queues (Q) and stacks (S), $I \in \{F, B\}$ (front, back) is the position of inserts in the list, and $D \in \{F, B\}$ is the position of deletes in the subscript X_{ID}.

Test Program	$\|\mathcal{R}\|$	Potential Ops		Labelled Ops				Data Structures		
		$\|\mathcal{P}\|$	$\frac{\|\mathcal{P}_\mathcal{R}\|}{\|\mathcal{R}\|}$	$\frac{\|\mathcal{L}_\mathcal{R}\|}{\|\mathcal{P}_\mathcal{R}\|}$	$\frac{\|\mathcal{L}_\mathcal{R}\|}{\|\mathcal{R}\|}$	$1-\frac{\|\mathcal{L}_\mathcal{R}\|}{\|\mathcal{L}\|}$	$\mathcal{L}_\text{quality}$	Label	Correct	% of $\mathcal{L}$ supports
Wolf SLL	200	468.1	0.97	0.94	0.91	0.03	1.00	SLL	✓	100%
Weiss SLL	200	445.4	0.93	0.73	0.69	0.13	0.99	SLL^H	✓	100%
Wolf DLL*	200	450.8	1.00	0.35	0.35	0.57	0.98	DLL	✓	100%
Wolf DLL 2	200	405.2	0.95	0.84	0.80	0.00	1.00	DLL	✓	100%
Wolf Stack	200	529	0.88	0.83	0.73	0.00	0.92	S^H_{FF}	✓	100%
Sedgewick Stack	200	522	0.85	0.79	0.68	0.00	0.93	S_{FF}	✓	100%
Weiss Stack	200	418.1	0.80	0.68	0.56	0.00	0.96	S^H_{FF}	✓	100%
Wolf Queue	200	434.4	0.85	0.77	0.67	0.05	0.95	Q^H_{BF}	✓	95%
Deshpande Tree	300	759.3	1.10	0.61	0.67	0.11	0.91	bTree	✓	91%
SLL Perm	300	395	0.96	0.73	0.70	0.06	0.89	SLL	✓	100%
DLL Perm	300	382	0.95	0.69	0.66	0.08	0.91	DLL	✓	100%
Multiple DSs	800	868.5	1.04	0.70	0.73	0.04	0.96	N/A		
mp3reorg†	111.2	111.8	0.98	0.99	0.97	0.00	0.58	SLL	✓	100%
Acidblood†*	439.8	804.8	0.77	0.71	0.55	0.03	0.67	DLL	✓	100%
Olden Health*	92.9	504.7	0.76	0.51	0.40	0.20	0.68	DLL	✓	86%

$$\mathcal{L}_\text{quality} = \frac{1}{|\mathcal{L}|} \sum_{L \in \mathcal{L}} \left(\frac{|L \cap \psi(L)|}{|\psi(L)|} - \frac{|L| - |L \cap \psi(L)|}{|L|} \right)$$

We report the following quantities to assess our approach. Firstly, we give $\frac{|\mathcal{P}_\mathcal{R}|}{|\mathcal{R}|}$, the fraction of operations that correctly locate a real operation (this may be greater than 1 due to the loose classification of a "correct" potential operation). $|\mathcal{P}_\mathcal{R}|$ imposes an upper bound on the success of the labelling, as we may only label potential operations. Thus, we report $\frac{|\mathcal{L}_\mathcal{R}|}{|\mathcal{P}_\mathcal{R}|}$, which is the quality of the labelling wrt. the potential operations. $\frac{|\mathcal{L}_\mathcal{R}|}{|\mathcal{R}|}$ is the overall success of the approach in identifying operations. The false-positive (FP) rate is given by $1 - \frac{|\mathcal{L}_\mathcal{R}|}{|\mathcal{L}|}$, i.e., the fraction of incorrectly labelled operations. Lastly, we determine the data structure label and report the percentage of $\mathcal{L}$ supporting this choice.

Results. The results for our technique are presented in Table 2. When interpreting these, we must keep in mind our goal, namely to be able to classify data structures based on the operations that manipulate them. Therefore, while the overall number of operations that are correctly labelled is important, the FP rate is just as important. In other words, a low FP rate and a reasonable number of correctly labelled operations gives strong evidence for the label of a data structure. It is important to note that the FP rate is reported *in terms of* *labelled operations.* For example, in Wolf SLL, on average 91% of operations are correctly labelled, and only 3% of the 91% are false positives.

Our tool identifies the following operation categories: SLL/DLL Insert/Remove Front/Middle/Back and Tree Insert/Remove. The position element is mandatory to identify data structures such as queues and stacks when implemented using lists, as our textbook examples are; thus, we must also identify when a list uses a header node. For all examples, the type of the data structure being manipulated is correctly inferred. Obviously, there is some overlap between queues, stacks and SLLs; however, by preferring the label of the more restrictive data structure when the evidence is within a tolerance of the other possibilities ($t_{\text{ops}} = 10\%$ in our experiments), the correct label is easily inferred. For example, in Wolf Queue, 100% of operations support SLL and 95% support Q^H_{BF}; since Q^H_{BF} is the more restrictive label and within the tolerance, this label is chosen.

In general, the fraction of operations correctly labelled is high, and the FP rates are low. The FP rates for lists, stacks and queues are all explained by the operation position being incorrectly identified. This is indeed the case for Wolf DLL, where a tail pointer makes the shape symmetric and causes the position to be incorrectly identified. When this experiment is re-run without requiring correct position (Wolf DLL 2), the results are much improved. We discuss solutions to these types of problems in the next section. The only examples to have operations that oppose the chosen labelling are Wolf Queue and Deshpande Tree. As elements are always inserted to the back of the queue, these negative examples arise when inserting to an empty queue, and hence, an Insert Front operation is recognized. For Deshpande Tree, the operations are coded iteratively and, therefore, display many modes of execution; some of the patterns inadequately cover the operations and cause an operation to be identified as an SLL operation.

In SLL Perm and DLL Perm we correctly permute insert and delete operations to check the robustness of our approach against various coding styles. Four variants are tested, and these can all be recognized with a low FP rate.

Program Multiple DSs uses an SLL, a DLL, a cyclic DLL and a binary tree together, where each data structure maintains a sorted set of integers. Repeatedly, the program randomly chooses a data structure, and randomly chooses an insert or delete operation to perform. The recognition rates are high and FP rates are low, showing that the combination of templates and attributes provides good discrimination of operations. The only overlap that occurs is in corner cases, such as inserting into an empty tree or DLL. The different data structures all use the same type (except SLL), so discrimination based on these is impossible. We do not require the position to be correctly identified for this test.

Program mp3reorg [11] is a small open-source program ($\approx$ 450 LOC) for organizing the layout of mp3 files from their ID3 tags. We vary the mp3s in the input directory to obtain multiple runs. The trace contains noise in the form of pointers for handling files, and the list elements have pointer payloads of malloc'ed strings, thus confusing the set of difference vertices. Nevertheless, we achieve very good recognition rates for this program.

Acidblood [1] is a medium-sized open-source program (≈ 5k LOC) implementing an IRC bot. It uses 14 different user-defined `struct`s for servers, commands, users, networking, etc., and some of these represent linked lists. We allow it to connect to a server and then simulate privileged users being randomly added and removed. Traces from this program include much noise in the form of pointer writes for network management. Furthermore, some `struct`s contain many fields, meaning that preparing the payload is a significant portion of an insert. Our tool can recognize a significant number of the DLL operations correctly, and thus, we can accurately infer the data structure used.

Program Olden Health (≈ 500 LOC) gives the results for the program *health* from the Olden benchmark [13]. This program contains much noise and has nested data structures, so we are operating at the limit of our approach here. However, despite a slightly low recognition rate, the type of the list data structure being used can be correctly determined as a DLL.

This proves that our prototype tool is successful at identifying data structures based on the set of operations that manipulate them. Typically, operation labels have a low FP rate, showing that the probability of a data structure being assigned an incorrect label is small.

6 Related Work and Conclusions

Discovering Data Structures. The shape of a data structure is commonly abstracted by a shape graph, which permits finite representations of unbounded recursive structures. These may be discovered for profiling and optimization [15], detecting abnormal data structure behaviour [9] and constructing program signatures [2]. In contrast, the whole heap is modelled in [14] to capture the dynamic evolution of data structures in Java programs and is used to collect summary statistics, including tree/DAG/cycle classification (a classification similar in scope to the static approach of [5]). However, none of these approaches consider the operations affecting the data structures and, hence, fail to capture dynamic properties such as linked lists implementing queues.

DDT [10] is the closest to our approach and functions by exploiting the coding structure in standard library implementations to identify interface functions for data structures. Invariants are then constructed, describing the observed effects of an interface function, and these are used in turn to classify the data structures. The reliance on well-structured interface functions means the approach is not designed for the customised interfaces appearing in OS/Legacy Software and C programs, or the replicated interfaces that appear due to function inlining. In contrast, our machine learning approach makes fewer assumptions about the structure of the code implementing operations.

Verifying Data Structure Usage. Today, shape analysis [17] is one of the predominate ways to reason about heap pointer structures. This framework is based on static analysis and enables the automated proof of program properties that relate to the shape of data structures on the heap. To configure the framework via custom predicates, some information on the shape of the data structure under analysis must be known *a priori*, although there exists some work on inferring predicates automatically [7]. The situation is similar for the proofs carried out in separation logic [16] and abstraction-based techniques such as [8], where abstractions need to be tailored to the data structures at hand.

Conclusions and Future Work. We presented an approach for learning the data structure operations employed by a pointer program given only an execution trace. Our evaluation on a prototypic implementation showed that the false-positive rate is low, and thus, the labelled operations can accurately infer the data structures they manipulate. We wish to apply this work to various domains, including automated verification, program comprehension and reverse engineering, and to make our prototype available after having been generalised wrt. nested data structures, non-tail-recursive operations and object code analysis. Last but not least, we wish to thank the anonymous reviewers for their valuable comments and suggestions.

References

1. Acidblood IRC Bot, `freecode.com/projects/acidblood` (accessed September 30, 2012)
2. Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging for data structures. In: OSDI, pp. 255–266. USENIX (2008)
3. Deshpande, P.S., Kakde, O.G.: C & Data Structures. Charles River (2004)
4. Evolving Objects (EO), `eodev.sourceforge.net` (accessed September 30, 2012)
5. Ghiya, R., Hendren, L.J.: Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL, pp. 1–15. ACM (1996)
6. Grünwald, P.D.: The Minimum Description Length Principle. MIT Press (2007)
7. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: PLDI, pp. 256–265. ACM (2007)
8. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph grammar abstraction for unbounded heap structures. ENTCS 266, 93–107 (2010)
9. Jump, M., McKinley, K.S.: Dynamic shape analysis via degree metrics. In: ISMM, pp. 119–128. ACM (2009)
10. Jung, C., Clark, N.: DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage. In: MICRO, pp. 56–66. ACM (2009)
11. MP3 File Reorganizer, `sourceforge.net/projects/mp3reorg` (accessed September 30, 2012)
12. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
13. Olden Benchmark, `www.martincarlisle.com/olden.html` (accessed September 30, 2012)
14. Pheng, S., Verbrugge, C.: Dynamic data structure analysis for Java programs. In: ICPC, pp. 191–201. IEEE (2006)
15. Raman, E., August, D.I.: Recursive data structure profiling. In: MSP, pp. 5–14. ACM (2005)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE (2002)

17. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS 24(3), 217–298 (2002)
18. Sedgewick, R.: Algorithms in C – Parts 1-4, 3rd edn. Addison-Wesley (1998)
19. Weiss, M.A.: Data structures and algorithm analysis in C. Cummings (1993)
20. Wolf, J.: C von A bis Z. Galileo Computing (2009)

Synthesis of Circular Compositional Program Proofs via Abduction*

Boyang Li[1], Isil Dillig[1], Thomas Dillig[1], Ken McMillan[2], and Mooly Sagiv[3]

[1] College of William & Mary
[2] Microsoft Research
[3] Tel Aviv University

Abstract. This paper presents a technique for synthesizing circular compositional proofs of program correctness. Our technique uses abductive inference to decompose the proof into small lemmas, which are represented as small program fragments annotated with pre and post-conditions. Different tools are used to discharge each different lemma, combining the strengths of different verifiers. Furthermore, each lemma concerns the correctness of small syntactic fragments of the program, addressing scalability concerns. We have implemented this technique and used it combine four different verification tools. Our experiments show that our technique can be successfully used to verify applications that cannot be verified by any individual technique.

1 Introduction

Different program verifiers have different limitations. For example, some may fail to prove a property because they use a coarse abstraction of the program semantics. In this category, we find abstract interpreters and verification condition generators, which require the property to be proved to be inductive. Others model the program semantics precisely, but often do not scale well in practice. In this category, we find model checkers and inductive invariant generators. To accomodate the limitations of program verifiers, a classical approach is synthesizing *compositional proofs*. The idea is to decompose the correctness proof of the program into a collection of lemmas, each of which can be verified by considering a small syntactic fragment of the program. This directly addresses the question of scalability, and indirectly the question of abstraction, since each lemma may be provable using a fairly coarse abstraction, even if the overall property is not.

The key difficulty in synthesizing compositional proofs is to discover a suitable collection of lemmas. Automating this process has proven to be extremely challenging. Some progress has been made in the finite state case [1,2] and in some particular domains such as shape analysis [3]. However, general approaches for inferring compositional proofs are lacking.

In this paper, we describe an approach to inferring lemmas based on logical abduction, the process of inferring premises that imply observed facts. Specifically, our technique uses abduction to synthesize *circular compositional proofs*.

* This work is supported in part by DARPA #FA8750-12-2-0020.

```
1.  int i=1; int j=0;
2.  while(*) { j++; i+=3; }
3.  int z=i-j;
4.  int x=0; int y=0; int w=0;
5.  while(*) [assert(x=y)]
6.  { z+=x+y+w; y++; x+=z%2; w+=2; }
```

Fig. 1. Example to illustrate main ideas of our technique

In such a proof, each lemma is a fact that must hold at all times, and we must prove that each lemma is not the first to fail. In effect, the proof of each lemma is allowed to assume the correctness of all the others, the apparent circularity being broken by induction over time. Our goal is to introduce lemmas that can be discharged in this way, using only small program fragments.

A key feature of our approach is that it is lazy. That is, when a lemma $\mathcal{L}$ cannot be discharged, our technique introduces a new lemma that may help to prove $\mathcal{L}$. The key insight is that such useful auxiliary lemmas can be inferred by combining verification condition (VC) generation with logical abduction [4]. Specifically, given an invalid VC $\phi_1 \Rightarrow \phi_2$, we employ abductive inference to infer an auxiliary lemma ψ such that $\psi \wedge \phi_1 \Rightarrow \phi_2$ is valid. Experimentally, we observe that lemmas generated to help verification condition checking are also useful for other types of verifiers, such as model checkers and abstract interpreters.

The ability to synthesize compositional proofs by inferring relevant lemmas has two important benefits. First, it helps us to address the problems of scale and abstraction. The lemmas can be verified on small program fragments, and each can be checked using a different abstraction. Second, lemmas allow us to combine the strengths of many verifiers, as each lemma may be verified by a different tool. The tools can be used as black boxes, without any modification.

This paper applies these ideas for verifying safety properties of sequential programs. In principle, though, they can be applied to any class of programs and any proof system generating verification conditions in a suitable form.

1.1 Overview

Given an imperative program containing *assume* and *assert* statements, we want to show that no assertion fails in any execution. Our safety proof makes use of two basic steps: introduction and elimination of assertions. In an introduction step, we insert a new assertion at any point in the program. In an elimination step, we prove that some assertion always holds and then convert it to an assumption. When verifying an assertion $\mathcal{A}$, we can convert all the other assertions to assumptions, since we are only proving that $\mathcal{A}$ is not the first to fail. Moreover, given these assumptions, we might be able to verify our assertion locally, using some small fragment of the program containing the assertion.

As an example, consider the program of Figure 1. The assertion in square brackets on line 5 represents an *invariant* of the loop. It must hold each time the loop is entered and also when the loop exits. We would like to verify this invariant assertion using just lines 4–6 in isolation. This is not possible, however, because we require the precondition "z is odd" established by lines 1–3. Having failed in our verification attempt, we will try to infer a lemma that makes the verification possible. For this, we decorate the program with symbols representing unknown assumptions. We then compute a *verification condition* (VC), that is, a logical formula whose validity implies the correctness of the decorated program. Then, using a technique known as *abduction*, we will solve for values of the unknown assumptions making the VC valid. These assumptions will then become lemmas to be proved. Going back to our example, we decorate lines 4–6 as follows:

```
4.    int x=0; int y=0; int w=0;
      assume $\phi_1$
5.    while(*) [assert x=y; assume $\phi_2$]
6.            z+=x+y+w; y++; x+=z%2; w+=2;
```

The symbols ϕ_1 and ϕ_2 are placeholders for unknown assumptions. The assumption ϕ_1 is a precondition for the loop, while ϕ_2 is an additional (assumed) invariant. Our VC generator tells us that our decorated program is correct when the following formulas are valid:

$$(z = i - j \land x = 0 \land y = 0 \land w = 0 \land \phi_1) \Rightarrow x = y$$
$$(\phi_2 \land x = y) \Rightarrow \mathrm{wp}(\sigma, x = y)$$

Here, σ is the loop body (the code of line 6), and $\mathrm{wp}(\sigma, \phi)$ stands for the weakest liberal precondition of formula ϕ with respect to statement σ. These conditions say that the invariant $x = y$ must hold on entering the loop, and that it is preserved by the loop body, given our assumptions.

Now, we can easily see that the first condition is valid, but the second one is not valid. Using the definition of wp, the second condition is equivalent to:

$$(\phi_2 \land x = y) \Rightarrow x + (z + x + y + w)\%2 = y + 1$$

To prove the invariant $x = y$, we need to find a formula to plug in for ϕ_2 that makes this formula valid. At the same time, we do not want our new lemma ϕ_2 to contradict the original lemma $x = y$ that we are trying to prove. Thus, we want $\phi_2 \land x = y$ to be satisfiable. This problem of inferring a hypothesis that implies some desired fact, while remaining consistent with given facts, is known as *abduction*. Using the algorithm described in Section 4, we obtain the solution $(w + z)\%2 = 1$ for this abduction problem.

Having inferred an auxiliary invariant $(w + z)\%2 = 1$ through abduction, this formula now becomes a lemma in our proof. We *introduce* the invariant assertion "assert $(w + z)\%2 = 1$", so lines 4–6 now look like this:

```
4.    int x=0; int y=0; int w=0;
5.    while(*) [assert x=y; assert (w+z)%2 = 1]
6.            z+=x+y+w; y++; x+=z%2; w+=2;
```

We can now prove the assertion $x = y$ by assuming our new lemma. We therefore *eliminate* this assertion by converting it to an assumption, obtaining:

```
4.    int x=0; int y=0; int w=0;
5.    while(*) [assume x=y; assert (w+z)%2 = 1]
6.              z+=x+y+w; y++; x+=z%2; w+=2;
```

Unfortunately, the lemma $(w + z)\%2 = 1$ still cannot be proved using just these code lines, since it depends on the initial value of z, which is determined by the first loop. Therefore, we once again decorate the program with unknown assumptions ϕ_1 and ϕ_2. The VC's of the new program are:

$$(z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge \phi_1 \wedge x = y) \Rightarrow (w + z)\%2 = 1$$
$$\phi_2 \wedge (w + z)\%2 = 1 \wedge x = y \Rightarrow \text{wp}(\sigma, x = y \Rightarrow (w + z)\%2 = 1)$$

where again σ is the loop body. That is, our lemma must hold on entry to the loop, and must be preserved by the loop, given our assumptions. However, neither of these conditions is valid, so we try to repair the first condition. To make it valid, we need to find a formula ψ to plug in for ϕ_1 such that:

$$(\psi \wedge z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge x = y) \Rightarrow (w + z)\%2 = 1$$
$$(\psi \wedge z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge x = y) \not\Rightarrow \textit{false}$$

That is, the assumption ψ must be sufficient to establish the invariant on entry to the loop, but not contradict known facts, including the invariant $x = y$. Our abduction technique discovers the solution $z\%2 = 1$ for ψ.

This solution $z\%2 = 1$ for ϕ_1 now becomes a lemma, introduced as an assertion before the loop. We now have:

```
4.    int x=0; int y=0; int w=0;
      assert z%2 = 1;
5.    while(*) [assume x=y; assert (w+z)%2 = 1]
6.              z+=x+y+w; y++; x+=z%2; w+=2;
```

At this point we have two assertions in the program. The VC for the loop invariant is still not valid (that is, the invariant is not inductive). However, at this point we *can* verify it using just lines 4–6 in isolation, since we have the necessary precondition $z\%2 = 1$. Converting this assertion to an assumption, we give the above fragment to a client program analyzer. If this client tool is able to infer divisibility facts, it can verify the invariant by inferring the auxiliary invariant $w\%2 = 0$. We have therefore *localized* the verification of the loop invariant.

Having verified the assertion $(w + z)\%2 = 1$, we eliminate it by converting it to an assumption and we move on to the remaining assertion, $z\%2 = 1$. This assertion can be verified using lines 1–4 in isolation. That is, we give these lines to a client program analyzer that is able to infer the linear invariant $i = 3j + 1$ of the first loop. From this, it can prove that z is odd. All assertions have now been eliminated, so the program is verified.

Notice that our inference of lemmas using abduction had two significant advantages in this example. First, it allowed us to *localize* the verification, proving one lemma using just the first loop, another one using just the second. This addresses the issue of scale. Second, we were able to verify these lemmas using two different *abstractions*, in one case using divisibility predicates, and the other using linear equalities. In this way, proof decomposition allows different program verification tools to be combined as black boxes.

2 Language and Preliminaries

In this section, we give a small language on which we formalize our technique:

$$\begin{array}{lll}
\text{Program } Pr & := & s \\
\text{Statement } s & := & \texttt{skip} \mid v := e \mid s_1; s_2 \mid \texttt{if}(\star)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
 & & \mid \texttt{while}(\star)[s_1]\ \texttt{do}\ \{s_2\} \mid \texttt{assert}\ p \mid \texttt{assume}\ p \\
\text{Expression } e & := & v \mid c \mid e_1 + e_2 \mid e\%c \mid c * e \\
\text{Predicate } p & := & e_1 \oslash e_2 \quad (\oslash \in \{<, >, =\}) \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p
\end{array}$$

A program consists of one or more statements. Statements include skip, assignments, sequencing, if statements, while loops, assertions, and assumptions. While loops may be decorated with invariants using the $[s]$ notation. The code s is executed before the loop body and also before exiting the loop, and may contain assert and assume statements. Expressions include variables, constants, addition, multiplication, and mod expressions. Predicates are comparisons between expressions as well as conjunction, disjunction, and negation.

We assume a scheme for numbering the statements in a program, including compound statements. Given a program π and a statement number (or *position*) p occurring in π, we write $\pi|_p$ for the statement in π numbered p. Moreover, given a statement σ, we write $\pi[\sigma]_p$ for π with σ replacing the statement numbered p. We also use $\text{asrts}(\pi)$ to represent the set of positions of assert statements in π and $\text{elim}(\pi, P)$, where P is a set of assert positions, to represent π with all asserts in positions P converted to assumes. The notation $\text{elim}(\pi, \neg p)$ is a shorthand for $\text{elim}(\pi, \text{asrts}(\pi) \setminus \{p\})$, that is, π with all asserts *except* position p converted to assumes. We use $\text{elim}(\pi)$ for π with all asserts converted to assumes.

3 Searching for Circular Compositional Proofs

In our proofs, we use a vocabulary Σ_U of placeholder symbols to stand for unknown program invariants. A placeholder $\phi \in \Sigma_U$ may occur only in a statement of the form "assume ϕ". We also use an operator spr that, given a program π, returns a formula whose validity implies correctness of π. That is, $\models \text{spr}(\pi)$ implies $\models \text{wp}(\pi, \mathit{true})$. The operator spr is, in effect, our VC generator. We assume that our VC generator spr returns a set of clauses of the form:

$$\chi \wedge \phi_p \Rightarrow \Gamma$$

$$\frac{\vdash \pi[\text{assert } \psi;\ \sigma]_p}{\vdash \pi[\sigma]_p}\ \textsc{Intro} \qquad \frac{\begin{array}{l}\vdash \text{elim}(\pi, \neg p)\\ \vdash \text{elim}(\pi, p)\end{array}}{\vdash \pi}\ \textsc{Elim} \qquad \frac{\begin{array}{l}\vdash \sigma\\ \vdash \pi[\text{elim}(\sigma)]_p\end{array}}{\vdash \pi[\sigma]_p}\ \textsc{Localize}$$

Fig. 2. Inference rules for compositional proof

where $\phi_p \in \Sigma_U$. The *constraint* χ does not contain placeholders, and the *goal* Γ is some formula asserted in the program. We also allow placeholder-free clauses of the form $\chi \Rightarrow \Gamma$. Our VC generation scheme (Section 3.3) is designed to produce VC's in these forms.

Our proof search algorithm makes use of three proof rules shown in Figure 2. These rules produce judgements of form $\vdash \pi$, where π is a program. The meaning of this judgement is that π does not fail in any context, i.e., $\text{wp}(\pi, \mathit{true}) = \mathit{true}$.

Rule Intro allows us to insert a new assertion in any syntactic position in the program. This rule is sound because adding an assertion can only strengthen the weakest precondition. The Intro rule is used in our proof search algorithm to introduce auxiliary lemmas in the form of assertions in the source code.

Rule Elim allows us to eliminate an assertion that is true. It says that, if the program is correct with all assertions *except p* converted to assumes, then we can convert p to an assume. Effectively, the Elim proof rule justifies the use of circular compositional reasoning in our approach. This rule will be useful in our proof search algorithm because it says that we can assume the correctness of all other assertions in proving the correctness of assertion p.

Finally, the Localize rule allows us to syntactically localize the verification of an assertion. That is, if a fragment of the program containing assertion p is correct, then p is correct in the entire program. This rule allows us to decompose large programs into smaller syntactic components for verification. The leaf subgoal $\vdash \sigma$ in this rule will be discharged by an oracle, which is our set of program verifiers. If the oracle certifies that σ is correct, then we take $\vdash \sigma$ as an axiom.

In searching for a proof in this system, we must make a number of heuristic decisions. For example, we must decide in what order to process subgoals, and, at each subgoal, we must choose a proof rule to apply. When applying the Intro rule, we must choose where and what assertions to introduce. Similarly, for Elim, we must choose the order of elimination of assertions, and for Localize, we must decide what program fragment σ to use for the verification of an assertion. Moreover, if a subgoal is unprovable (for example, because we introduced an assertion that is not correct), then we require a backtracking strategy.

Our tactic for searching for a proof in this system is illustrated in pseudocode in Figure 3. To reduce clutter, we don't construct the actual proof. Instead we just return *true* if a proof of the goal $\vdash \pi$ is found. We start by choosing an arbitrary assertion p to eliminate using the Elim rule (line 3). We call procedure Localize (line 4) to produce a local fragment for verifying p, using the Localize rule. In our implementation we use the inner-most while loop σ

```
Procedure ProofSearch($\pi$):
    input: program $\pi$
    output: true if proof of $\pi$ succeeds

    (1)  let $P = \text{asrts}(\pi)$
    (2)  if $P$ is empty, return true
    (3)  choose some $p \in P$, and let $\pi' = \text{elim}(\pi, \neg p)$
    (4)  let $\sigma = \text{Localize}(\pi', p)$
    (5)  if the oracle certifies $\sigma$ or $\models \text{spr}(\pi')$ then
    (6)      return ProofSearch(elim($\pi, p$))
    (7)  let $\mathcal{I} = \text{InferByAbduction}(\pi')$
    (8)  for each $(p', \phi)$ in $\mathcal{I}$ do
    (9)      let $\pi'' = \pi[\text{assert } \phi;\ \pi|_{p'}]_{p'}$
    (10)     if ProofSearch($\pi''$) then return true
    (11) done
    (12) return false
```

Fig. 3. Proof search algorithm

containing p. We then ask the oracle to prove the assertion (including a VC check). If the oracle can prove σ, we move on to the remaining assertions by processing the second sub-goal of the Elim rule (line 6).

On the other hand, if the oracle fails, we use abduction to generate a sequence of possible lemma introductions in order to make p provable (line 7). We try these in turn, applying the Intro rule (line 9) and recurring on the generated subgoal (line 10). If this proof fails, we move on to the next lemma in the sequence, and so on, until the sequence is exhausted, at which point, we return failure.

3.1 Using Abduction to Infer New Assertions

The key step in our proof search algorithm is the InferByAbduction procedure, shown in Figure 4. This procedure takes a program π and suggests new assertions that may be introduced to help make π provable. The first step in this process is to decorate the program with some assumptions of the form "assume ϕ_p", where ϕ_p is a placeholder symbol corresponding to statement position p. These placeholders stand for possible assertions we could introduce in a compositional proof. We discuss the choice of the placeholder locations in Section 3.2.

The next step is to generate the VC for the decorated program using the spr operator (described in Section 3.3). This is a set of clauses of the form $\chi \Rightarrow \Gamma$ or $\chi \wedge \phi_p \Rightarrow \Gamma$. To prove the assertion, we need to choose values of the placeholders to make all of these implications valid. If there is an invalid clause of the form $\chi \Rightarrow \Gamma$ we cannot succeed, so we return the empty sequence. Otherwise, we consider each invalid clause of the form $\chi \wedge \phi_p \Rightarrow \Gamma$. We want to choose a formula to assign to ϕ_p in order to make the implication $\chi \wedge \phi_p \Rightarrow \Gamma$ valid. In addition, we do not want the implication to be vacuously true, thus, we require that $\chi \wedge \phi_p$ be consistent.

```
Procedure InferByAbduction($\pi$):
    input: program $\pi$
    output: lazy list of pairs $(p, \phi_p)$

    let $\pi' = $ Decorate($\pi$)
    let VC = spr($\pi$)
    if there exists an invalid clause $\chi \Rightarrow \Gamma$ in VC then return
    for each invalid clause $\chi \wedge \phi_p \Rightarrow \Gamma$ in VC do
        for each $\psi$ in Abduc($\chi, \Gamma$) do
            yield $(p, \psi)$
        done
    done
```

Fig. 4. Inferring assertions by abduction

This leaves us with the following abduction problem. We must find a formula ψ over the program variables, such that the following two conditions hold:

$$\models \chi \wedge \psi \Rightarrow \Gamma \qquad \text{and} \qquad \not\models \chi \wedge \psi \Rightarrow \mathit{false}$$

In Section 4, we describe a method of solving this problem. For now, we assume a procedure Abduc that, given χ and Γ, returns a lazy list of solutions for ψ. InferByAbduction then returns the list of solutions $\psi_1, \psi_2, \ldots, \psi_n$ for each placeholder ϕ_p, paired with the corresponding program position p of ϕ_p.

3.2 Program Decoration

An important consideration in choosing the placement of placeholder assumptions is that each clause in the VC should contain a placeholder to allow us to to make progress when the VC is not valid (except, of course, for the whole program's precondition, which must be valid). In general, this placement strategy depends on the VC generation scheme. In our particular language and VC scheme, it suffices to put a placeholder at the head of each loop. To support localization (as seen in the example of Figure 1) we also add a placeholder before each loop. That is, the procedure Decorate replaces each statement of the form while$(\star)[\sigma]\{\tau\}$ in a program with:

```
assume $\phi_{\text{pre}}$;
while($\star$)    [$\sigma$; assume $\phi_{\text{inv}}$] { $\tau$ }
```

As a heuristic matter, we consider introducing a precondition for a loop before introducing an invariant.

3.3 VC Generation

The general approach we have described can use any VC generator function spr, provided the VC's can be rewritten into the required form. Here, we present a

$$(1)\frac{}{P, Q \vdash \texttt{skip} : \mathit{true}, P, Q} \qquad (2)\frac{Q' = \exists v'.(P[v'/v] \wedge v = (e[v'/v]))}{P, Q \vdash v := e : \mathit{true}, Q', Q[e/v]}$$

$$(3)\frac{Q' = P \wedge C \quad P' = Q \wedge C}{P, Q \vdash \texttt{assert}\ C : \mathit{true}, Q', P'} \qquad (4.1)\frac{\begin{array}{c}Q' = P \wedge C \quad P' = (C \Rightarrow Q)\\ C \text{ not placeholder}\end{array}}{P, Q \vdash \texttt{assume}\ C : \mathit{true}, Q', P'}$$

$$(4.2)\frac{\mathrm{VC}' = (P \wedge \phi_p(\boldsymbol{v}) \Rightarrow Q)}{P, Q \vdash \texttt{assume}\ \phi_p(\boldsymbol{v}) : \mathrm{VC}', \mathit{true}, \mathit{true}} \qquad (5)\frac{\begin{array}{c}P, P' \vdash s_1 : \mathrm{VC}_1, Q', P''\\ Q', Q \vdash s_2 : \mathrm{VC}_2, Q'', P'\end{array}}{P, Q \vdash s_1; s_2 : \mathrm{VC}_1 \wedge \mathrm{VC}_2, Q'', P''}$$

$$(6)\frac{\begin{array}{c}P, \mathit{true} \vdash I : \mathrm{VC}_1, \_, Q'\\ \mathit{true}, \mathit{true} \vdash \mathrm{elim}(I); s; I \ : \ \mathrm{VC}_2, \_, Q_2\\ \mathit{true}, Q \vdash \mathrm{elim}(I) \ : \ \mathrm{VC}_3, P', Q_3\\ \mathrm{VC}' = \mathrm{VC}_1 \wedge \mathrm{VC}_2 \wedge Q_2 \wedge \mathrm{VC}_3 \wedge Q_3\end{array}}{P, Q \vdash \texttt{while}(\star)[I]\ \texttt{do}\ \{s\} : \mathrm{VC}', P', Q'}$$

$$(7)\frac{\begin{array}{c}P, Q \vdash s_1 : \mathrm{VC}_1, Q_1, P_1 \qquad P, Q \vdash s_2 : \mathrm{VC}_2, Q_2, P_2\\ Q' = Q_1 \vee Q_2 \qquad P' = P_1 \wedge P_2\end{array}}{P, Q \vdash \texttt{if}(\star)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \mathrm{VC}_1 \wedge \mathrm{VC}_2, Q', P'}$$

Fig. 5. Rules describing computation of VC's

simple VC generation approach for programs without procedures that explicitly generates VC's in the form $\chi \wedge \phi_p \Rightarrow \Gamma$. The approach is based on propagating both strongest postconditions forwards and weakest preconditions backwards. However, we could also use a more standard approach based on just weakest preconditions with some rewriting of the result into the right form.

In our VC generation scheme, we generate a clause for each placeholder ϕ_p. Given the strongest postcondition of the code preceding p, this clause states that ϕ_p guarantees the weakest precondition of the code succeeding p. Since we can't compute preconditions and postconditions precisely for loops, we abstract these conditions, using the stated invariants of the loop. The result is a VC that is a sufficient but not necessary condition for the correctness of the program.

We describe our VC generation procedure as a set of inference rules (Figure 5) that produce judgements of the form $P, Q \vdash s \ : \ \mathrm{VC}', P', Q'$. The meaning of this judgement is that, if the environment of statement s guarantees precondition P and postcondition Q, then s will guarantee postcondition P' and precondition Q', given that VC' is valid. That is, the judgement is valid when $\models \mathrm{VC}'$ implies $\models P \Rightarrow \mathrm{wp}(s, P')$ and $\models Q' \Rightarrow \mathrm{wp}(s, Q)$.

For primitive statements s, we have $\mathrm{VC}' = \mathit{true}$, $P' = \mathrm{sp}(s, P)$ and $Q' = \mathrm{wp}(s, Q)$. Thus, our rules propagate strongest post-conditions forward and weakest pre-conditions backward. However, rule 4.2 is a special rule for placeholder assumptions. It produces a VC clause rather than propagating sp and wp.

For while loops (rule 6), we weaken the post-condition and strengthen the precondition by allowing entry to the loop in any state satisfying the stated loop invariants. The first premise guarantees that the loop invariant holds on entry, the second that the loop invariant is preserved by one iteration of the loop, and the third that exiting the loop satisfies its postcondition. One way to think of this is that, to verify a loop under pre- and post-conditions P and Q, we need to establish three Hoare triples: $\{P\}\ I\ \{true\}$ and $\{true\}\ elim(I); s; I\ \{true\}$ and $\{true\}\ elim(I)\ \{Q\}$. For example, in a typical case, we want to prove an invariant assertion ψ in a loop. The decorated loop looks like this:

$$\text{while}(\star) \quad [\text{assert } \psi; \text{ assume } \phi_{\text{inv}}] \qquad \{\ s\ \}$$

According to the first premise of rule (6), the precondition Q' of the loop is the precondition of "assert ψ; assume ϕ_{inv}", which is ψ. The postcondition P' of the loop (third premise) is the postcondition of "assume ψ; assume ϕ_{inv}", which is *true*, since ϕ_{inv} is a placeholder. Finally, the second premise yields the VC from:

$$\text{assume } \psi; \text{ assume } \phi_{\text{inv}};\ \ s;\ \ \text{assert } \psi; \text{ assume } \phi_{\text{inv}};$$

This yields two clauses, one for each placeholder instance, according to rule 4.2. The first is $\psi \wedge \phi_{\text{inv}} \Rightarrow \text{wp}(s, \psi)$. The second is *true*. To make the VC valid, we need to find an assumption ϕ_{inv}, under which ψ is inductive. Furthermore, since we add an "assume ϕ_{pre}" statement before the loop, Rule (4.2) results in the generation of the VC clause $P \wedge \phi_{\text{pre}} \Rightarrow \psi$ where P is the precondition of ϕ_{pre}. Thus, to make this VC valid, we must find an appropriate solution for ϕ_{pre} that implies ψ holds initially. Finally, the third premise of Rule (6) results in the generation of the VC $\psi \wedge \phi_{\text{inv}} \Rightarrow Q$, meaning that we must find a strengthening ϕ_{inv} of ψ that implies loop postcondition Q.

For program π, our goal is to derive a judgement of the form $true, true \vdash \pi\ :\ \text{VC}', \_, Q'$. This judgement says that if VC$'$ is valid, then a sufficient condition for correctness of our program in any initial state is Q'. Thus, we have $\text{spr}(\pi) = \text{VC}' \wedge Q'$. Using our particular decoration scheme, we are guaranteed that each clause in VC$'$ has exactly one occurrence of a placeholder (rule 4.2), or is free of placeholders (other rules).

Finally, we note that propagating postconditions forward has an additional advantage for compositional verification. That is, when we pass a localized program loop to the oracle for verification, we can include the precondition for that loop computed by our VC generator as an additional constraint on the initial state. This can allow us to verify assertions with smaller localizations.

4 Performing Abductive Inference

We now describe our technique for performing abductive inference, which corresponds to the ABDUC function used in the INFERBYABDUCTION algorithm. Recall that, given formulas χ and Γ, abduction infers a formula ψ such that:

$$(1)\ \chi \wedge \psi \Rightarrow \Gamma \qquad (2)\ \text{SAT}(\chi \wedge \psi)$$

While there are many formulas ψ that satisfy these two conditions, a *useful* abductive solution in our setting should have two characteristics:

1. First, ψ should contain as few variables as possible because invariants typically describe relationships between a few key variables in the program. For example, if both $x = y$ and $x + 10z + 5w - 4k \leq 10$ are sufficient to explain Γ, it is preferable to start with the simpler candidate $x = y$.
2. Second, ψ should be as general (i.e., as logically weak) as possible. For example, if $x = 0 \wedge y = 0$ and $x = y$ are both solutions to the inference problem, we prefer $x = y$ because solutions that are too specific (i.e., logically strong) are unlikely to hold for all executions of the program.

To find solutions containing as few variables as possible, observe that $\chi \wedge \psi \Rightarrow \Gamma$ can be rewritten as $\psi \Rightarrow (\neg\chi \vee \Gamma)$. Now, consider a satisfying assignment σ of $\neg\chi \vee \Gamma$ consistent with χ. By definition of a satisfying assignment, $\sigma \Rightarrow (\neg\chi \vee \Gamma)$. Thus, any satisfying assignment of $\neg\chi \vee \Gamma$ consistent with χ is a solution for the abductive inference problem. However, since we are interested in solutions with as few variables as possible, we are not interested in full satisfying assignments of $\neg\chi \vee \Gamma$, but rather *partial* satisfying assignments. Intuitively, a partial satisfying assignment σ of φ assigns values to a subset of the free variables in φ, but is still sufficient to make φ true, i.e., $\sigma(\varphi) \equiv$ *true*. Therefore, to find an abductive solution containing as few variables as possible, we will compute a *minimum partial satisfying assignment* (MSA) of $\neg\chi \vee \Gamma$ [5]. An MSA of formula φ is simply a partial satisfying assignment of φ containing no more variables than other partial satisfying assignments of φ. Minimum satisfying assignments for many theories, including Presburger arithmetic used in this paper, can be computed using the algorithm described in [5].

Now, if an MSA of $\neg\chi \vee \Gamma$ contains a set of variables V, we know there exists an abductive solution containing only V. However, we want to find a logically weakest formula over V that still implies $\neg\chi \vee \Gamma$. It can be shown that a weakest formula over V that implies $\neg\chi \vee \Gamma$ is given by $\forall\overline{V}.\ (\neg\chi \vee \Gamma)$ where $\overline{V} = \text{Vars}(\neg\chi \vee \Gamma) - V$. Furthermore, since we typically prefer quantifier-free solutions, quantifier elimination can be used to eliminate $\overline{V}$ in theories that admit quantifier elimination (such as Presburger arithmetic used here).

Example 1. Consider the problem from Section 1.1 of finding a ψ such that:

$$(1)\ \psi \wedge P \wedge x = y \Rightarrow wp(S, x = y) \qquad (2)\ \text{SAT}(\psi \wedge P \wedge x = y) \qquad \text{where}$$

$$\begin{aligned} P &= (z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0) \\ wp(S, x = y) &= (x + (z + x + y + w)\%2 = y + 1) \end{aligned}$$

To solve this problem, we first compute an MSA of $x \neq y \vee \neg P \vee wp(S, x = y)$ consistent with $P \wedge x = y$. Using the algorithm of [5], an MSA is $z = 1, w = 0$. Since variables x, y, i, j are not in the MSA, we generate the formula $\forall x, y, i, j.\ x \neq y \vee wp(S, x = y)$. Using quantifier elimination, this formula is equivalent to $(z + w)\%2 = 1$, which is the abductive solution we used in Section 1.1.

Name	LOC	Time (s)	# queries	Polyhedra	Linear Cong	Blast	Compass	Provable by RP?
B1	45	0.6	2	✘	✘	✔	✘	✘
B2	37	0.2	2	✘	✔	✘	✘	✘
B3	51	1.0	2	✔	✘	✔	✘	✔
B4	59	0.4	3	✔	✘	✔	✘	✘
B5	89	0.6	3	✔	✘	✔	✘	✘
B6	60	0.5	5	✘	✔	✘	✔	✘
B7	56	0.6	2	✘	✘	✔	✔	✘
B8	45	0.2	2	✔	✘	✔	✘	✔
B9	59	0.5	1	✘	✘	✔	✘	✘
B10	47	0.2	2	✔	✘	✔	✔	✘

Fig. 6. Experimental results on micro benchmarks

4.1 Computing All Abductive Solutions

In the previous discussion, we described how to compute one solution to the abductive inference problem defined by χ and Γ. However, the InferByAbduction algorithm from Section 3 requires a lazy list of solutions. That is, given a set of previous solutions $\psi_1, \psi_2, \ldots, \psi_k$ for the abduction problem defined by χ and Γ, how do we compute a new solution ψ_{k+1} distinct from $\psi_1, \psi_2, \ldots, \psi_k$?

To find such a solution ϕ_{k+1}, we compute an MSA of $\neg\chi \vee \Gamma$, that is not only consistent with χ but also with the *negations* $\neg\psi_1$, $\neg\psi_2$, ..., $\neg\psi_k$ of each of the previous solutions. Given such an MSA containing variables V, the formula $\forall\overline{V}.\ (\neg\chi \vee \Gamma)$ yields a new solution distinct from previous solutions. The process terminates when there is no longer a consistent solution.

5 Implementation and Experimental Evaluation

We have implemented the proposed technique using the SAIL front-end [6] for C programs and the Mistral SMT solver [7,5]. Mistral computes MSAs and performs quantifier elimination, which are necessary for performing abduction.

To evaluate our technique, we performed two experiments, one involving challenging synthetic benchmarks, and a second using open-source C programs. In both experiments, our oracle consists of four client tools: BLAST [8], the polyhedra abstract domain [9] implemented in the Interproc tool [10], the linear congruences domain [11] also implemented in Interproc, and Compass [12,13].

The results of the first experiment are summarized in Figure 6. This experiment involves 10 synthetic benchmarks available from http://www.cs.wm.edu/~tdillig/tacas-benchmarks.tar.gz. None of these benchmarks can be verified using one of the four client tools alone. Furthermore, even if we conjoin the invariants inferred by each tool, the combined invariants are still not sufficient to prove the assertion. However, using the proposed technique, all ten benchmarks can be verified using BLAST, polyhedra, linear congruences, and Compass as clients.

In Figure 6, the column labeled LOC shows the number of lines of code in each benchmark, and the column labeled “Time” shows analysis time in seconds, excluding the time taken by client tools to answer queries. The next column shows

Name	LOC	Time (s)	# queries	Avg # vars in query	Avg LOC in query
Wizardpen Linux Driver	1242	3.8	5	1.5	29
OpenSSH clientloop	1987	2.8	3	2.3	5
Coreutils su	1057	3.0	5	1.7	6
GSL Histogram	526	0.6	4	3.6	15
GSL Matrix	7233	16.9	8	1.8	7

Fig. 7. Experimental results on real benchmarks

the number of queries our technique poses to clients. The next four columns show which of the analyses were able to successfully answer at least one query on a given benchmark. Finally, the last column shows whether the original benchmark can be verified using the reduced product [14] of the convex polyhedra and linear congruences abstract domains, as implemented in Interproc.

The main point of the first experiment is that all benchmarks from Figure 6 can be verified using the proposed technique, although no client tool can individually verify any benchmark. Furthermore, the number of queries to client tools is small, ranging from 1-5 queries. This indicates that our technique is able to home in on relevant lemmas necessary to localize the overall proof. Figure 6 also shows that it is often helpful to combine different approaches in the verification task. For example, BLAST and polyhedra were useful for verifying benchmark 3, whereas linear congruences and Compass were used to verify benchmark 6.

In a second experiment, summarized in Figure 7, we used the proposed technique for verifying assertions in real C programs. The programs we analyzed include a Linux device driver, an OpenSSH component, a coreutil application, and two modules from the GNU scientific library (GSL). These benchmarks range from 526 to 7233 lines of code. As in the previous experiment, none of these benchmarks can be verified by individual client tools alone (i.e., they either do not terminate or report a false alarm). However, when the four client tools are combined using our technique, all benchmarks can be successfully verified.

Figure 7 also shows that, although the original programs are quite large, the extracted program fragments provided to client tools are small, ranging in size from an average of 5 to 29 lines. This corroborates the claim that our technique often extracts subgoals on program fragments that are much smaller than the original program. Although analyses like the polyhedra domain do not typically work on programs of this size, our technique can utilize such expressive analyses in the verification task by extracting small proof subgoals.

6 Related Work

Compositional Verification. The technique presented here is similar to other techniques for compositional verification such as [1,2,15]. Specifically, [1] and [2] use Angluin's L^* automata learning algorithm for learning assumptions in concurrent finite-state systems. In this work, we address synthesizing compositional proofs for sequential infinite-state systems, and our approach to generating missing assumptions is based on logical abduction rather than Angluin's learning algorithm. Similar to our proposed technique, the approach described in [15] also employs a circular compositional approach and uses different abstractions to discharge proof subgoals. However, in contrast to [15], our proof subgoals are generated automatically by abduction.

Combining Program Analyzers. Most previous work on combining verification tools focuses on abstract interpretation. Specifically, the reduced cardinal product [14] and logical product [16] constructions allow combining different abstract domains. Our work differs from these approaches in several respects: First, we do not require client tools to be based on abstract interpretation and treat each client tool as a black box. Second, our technique is compositional and does not require client tools to verify the entire program, but instead proof subgoals represented as small code snippets. This aspect of our technique allows utilizing very expensive analyses even when verifying large programs. Third, unlike the reduced product construction, our technique is automatic and does not need to be reimplemented for combining different analyses.

The Hector tool described in [17] also allows information exchange between different analysis tools. However, Hector does not generate proof subgoals, and information exchange is through first-order logic rather than source code.

Use of Abduction in Verification. Several other approaches have used abductive inference in the context of program verification [18,3,19]. Among these approaches, [3] and [19] also use abduction to generate missing preconditions. Specifically, [3] uses abduction for generating missing assumptions in an interprocedural shape analysis algorithm, whereas [19] uses abduction in the context of logic programming. Our work differs from [3,19] in that we address combining different verification tools in a compositional way and use a different algorithm for computing abductive solutions. Our own recent work also uses abductive inference to semi-automate the task of classifying error reports as false alarms or real bugs [20]. Similar to [20], we use minimum satisfying assignments [5] to solve abductive inference problems. However, the present work addresses the very different problem of combining different verification tools in one framework.

7 Conclusion

We have proposed an algorithm for automatically synthesizing circular compositional proofs of program correctness. Our technique employs logical abduction to infer auxiliary lemmas that are useful in a compositional proof. The inference of helper lemmas allows combining the strengths of different program verifiers in one framework, as different verifiers can be used to discharge different lemmas. We have implemented the proposed technique, and our experiments show that it can verify programs that cannot be proven by individual tools.

Acknowledgments. We would like to thank Hongseok Yang, Aaron Bradley, Peter O'Hearn, Noam Rinetzky, and the anonymous reviewers for their helpful feedback.

References

1. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning Assumptions for Compositional Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
2. Gupta, A., Mcmillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Form. Methods Syst. Des. (2008)
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, vol. 44(1), pp. 289–300 (2009)
4. Peirce, C.: Collected papers of Charles Sanders Peirce. Belknap Press (1932)
5. Dillig, I., Dillig, T., McMillan, K.L., Aiken, A.: Minimum Satisfying Assignments for SMT. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 394–409. Springer, Heidelberg (2012)
6. Dillig, I., Dillig, T., Aiken, A.: SAIL: Static Analysis Intermediate Language. Stanford University Technical Report
7. 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)
8. 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)
9. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. In: POPL, pp. 84–96. ACM (1978)
10. Jeannet, B.: Interproc analyzer for recursive programs with numerical variables, http://pop-art.inrialpes.fr/interproc/interprocweb.cgi
11. Granger, P.: Static Analysis of Linear Congruence Equalities Among Variables of a Program. In: Abramsky, S. (ed.) CAAP 1991 and TAPSOFT 1991. LNCS, vol. 493, pp. 169–192. Springer, Heidelberg (1991)
12. 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)
13. Dillig, I., Dillig, T., Aiken, A.: Precise reasoning for programs using containers. In: POPL (2011)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282. ACM (1979)
15. McMillan, K.L.: Verification of Infinite State Systems by Compositional Model Checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 219–237. Springer, Heidelberg (1999)
16. Gulwani, S., Tiwari, A.: Combining abstract interpreters. ACM SIGPLAN Notices 41, 376–386 (2006)
17. Charlton, N., Huth, M.: Hector: Software Model Checking with Cooperating Analysis Plugins (Tool Paper). In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 168–172. Springer, Heidelberg (2007)
18. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, pp. 235–246. ACM (2008)
19. Giacobazzi, R.: Abductive analysis of modular logic programs. In: Proceedings of the 1994 International Symposium on Logic programming, pp. 377–391. Citeseer (1994)
20. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI (2012)

As Soon as Probable: Optimal Scheduling under Stochastic Uncertainty

Jean-François Kempf, Marius Bozga, and Oded Maler

CNRS-VERIMAG
University of Grenoble
France
firstname.lastname@imag.fr

Abstract. In this paper we continue our investigation of stochastic (and hence dynamic) variants of classical scheduling problems. Such problems can be modeled as duration probabilistic automata (DPA), a well-structured class of acyclic timed automata where temporal uncertainty is interpreted as a bounded *uniform distribution* of task durations [18]. In [12] we have developed a framework for computing the expected performance of a *given* scheduling policy. In the present paper we move from *analysis* to *controller synthesis* and develop a dynamic-programming style procedure for automatically synthesizing (or approximating) *expected time optimal* schedulers, using an iterative computation of a *stochastic time-to-go* function over the state and clock space of the automaton.

1 Introduction

The allocation over time of reusable resources to competing tasks is a problem of almost universal importance, manifested in numerous application domains ranging from manufacturing (job shop) to program parallelization (task-graph). When the system admits uncertainty, for example, in task arrival time or duration, one cannot execute a fixed time-triggered schedule but needs to develop a scheduling *policy* (or *strategy*) that prescribes scheduling decisions at different *states* that the system may find itself in. The general problem falls into the scope of *controller synthesis* for timed systems [5] and its extensions toward time-optimality [4] and cost-optimality [14]. The reader is referred to [1] for a general framework for modeling and solving dynamic scheduling problems based on a restricted class of timed automata and to [17] for a more general exposition of control in the presence of adversaries.

The above mentioned work treats temporal uncertainty in the *set-theoretic* and *worst-case* tradition. A duration of a task can be *any* number in a given interval (no probability assigned) and the strategy is evaluated according to the worst performance induced by an instance of the external environment. Duration probabilistic automata (DPA) have been introduced in [18] to express stochastic scheduling problems and evaluate the expected performance of schedulers by methods similar to techniques used for generalized semi-Markov processes (GSMP) [10,11,7] such as [6,2]. This approach refines the successor operator used in the verification of timed automata [9,15] from a *zone transformer* into a *density transformer* and can compute, for example, the time distribution of reaching the final state and hence the expected termination time under a *given* scheduler. In [12] we developed an alternative clock-free procedure for evaluating the performance of schedulers by computing *volumes* of *zones* in the duration space.

In the present paper we move to the *synthesis* problem where we seek an optimal scheduling policy which decides in what global situations to *start* an enabled step of a process and under what conditions to *wait* and let other processes use the resource first. To this end we define a *stochastic time-to-go* function which assigns to any state of the schedule (global state of the automaton *and* the values of active clocks) the density of the time to total termination starting from this state and following the optimal strategy. These functions are piecewise-continuous and we show how they can be computed backwards from the final state.

The rest of the paper is organized as follows. Section 2 provides preliminary definitions. Section 3 introduces single processes, shows how to model them by automata and solve the (degenerate) optimal scheduling problem by backward value iteration. In Section 4 we introduce several processes running in parallel and admitting resource conflicts. We model them as products of automata and define schedulers that, in each state, resolve the non-determinism associated with the decision whether to start a step. In Section 5 we define the basic iterative step for computing the value (optimal expected time-to-go) in a state of the automaton based on the value of its successors. The value/policy iteration algorithm using these operators defines the optimal strategy. In Section 6 we study the computational aspects of the algorithm. First we characterize the class of time densities obtained by applying the passive race-analysis part of the iteration. Then we prove a non-laziness property of optimal schedulers for this class of problems, which facilitates the approximate solution of the optimization part of the iteration. A discussion of future work closes the paper.

2 Preliminaries

Definition 1 (Bounded Support Time Density). *A time density is a function* $\psi : \mathbb{R}_+ \to \mathbb{R}_+$ *satisfying*

$$\int_0^\infty \psi[t]dt = 1.$$

A time density is of bounded support when $\psi(t) \neq 0$ *iff* $t \in I$ *for some interval* $I = [a, b]$. *A partial time density satisfies the weaker condition:* $\int \psi[t]dt < 1$. *A bounded-support time density is uniform if* $\psi[t] = 1/(b - a)$ *inside its support* $[a, b]$.

We use a non-standard notation for *distributions*:

$$\psi[\leq t] = \int_0^t \psi[t']dt' \quad \psi[> t] = 1 - \psi[\leq t]$$

with $\psi[\leq t]$ indicating the probability of a duration which is at most t. We use $\mathbf{c}$ to denote the "deterministic" (Dirac) density which gives the constant c with probability 1. The *expected value* of a time density ψ is $\mathbb{E}(\psi) = \int \psi[t] \cdot tdt$.

We will use time densities to specify durations of tasks (process steps) as well as the remaining time to termination given some state of the system. To this end we need the following *operators* on densities: 1) *Convolution*, to characterize the duration of two or more tasks composed sequentially, and 2) *Shift*, to reflect the change in the remaining time *given* that the process has *already* spent some amount of time x.

Definition 2 (Convolution and Shift). *Let ψ , ψ_1 and ψ_2 be uniform densities supported by $I = [a, b]$, $I_1 = [a_1, b_1]$ and $I_2 = [a_2, b_2]$, respectively.*

- *The convolution $\psi_1 * \psi_2$ is a density ψ' supported by $I' = I_1 \oplus I_2 = [a_1 + a_2, b_1 + b_2]$ and defined as*
$$\psi'[t] = \int_0^t \psi_1[t']\psi_2[t - t']dt'$$
- *The residual density (shift) of ψ relative to a real number $0 \leq x < b$ is $\psi' = \psi_{/x}$ such that $\psi'[t] = \psi[x + t] \cdot \gamma_{a,b}(x)$ where*
$$\gamma_{a,b}(x) = \begin{cases} 1 & \text{if } 0 < x \leq a \\ \frac{b-a}{b-x} & \text{if } a < x < b \end{cases}$$

When $x < a$, $\psi_{/x}$ is a simple shift of ψ. When $x > a$ we already know that the actual duration is at least x (restricted to the sub-interval $[x, b]$) and hence we need to normalize. Note also that $(\psi * \mathbf{c})[t] = (\mathbf{c} * \psi)[t] = \psi[t - c]$ and that **0** is the identity element for convolution. We write $\psi' = \psi_{/x}$ more explicitly as

$$\psi'[t] = \begin{cases} 0 & \text{when } x + t \leq a \\ \frac{1}{b-a} & \text{when } x \leq a, a < x + t \leq b \\ \frac{1}{b-x} & \text{when } a < x, x + t \leq b \\ 0 & \text{when } b < x + t \end{cases}$$

One can verify that the shift satisfies: $\psi_{/x}[t-x] = \gamma_{a,b}(x) \cdot \psi[t]$ and $\left(\psi_{/x}\right)_{/y} = \psi_{/(x+y)}$. Note that the expressive weakness (and computational advantage) of the exponential distribution is due to $\psi_{/x} = \psi$. A subset of a hyper-rectangle is called a *zone* if it is expressible as a *conjunction* of *orthogonal* and *difference* constraints, namely constraints of the form $x_i \leq c$, $x_i - x_{i'} \leq c$, etc.

3 Processes in Isolation

Processes in our model are inspired by the jobs in the job-shop problem. Each process consists of an ordered sequence of steps, indexed by $K = \{1, \ldots, k\}$, such that a step can start executing only after its predecessor has terminated.[1]

Definition 3 (Process). *A sequential stochastic process is a pair $P = (\mathcal{I}, \Psi)$ where $\mathcal{I} = \{I_j\}_{j \in K}$ is a sequence of duration intervals and $\Psi = \{\psi_j\}_{j \in K}$ is a matching sequence of densities with ψ_j being the uniform density over $I_j = [a_j, b_j]$, indicating the duration of step j.*

Probabilistically speaking, step durations can be viewed as a finite sequence of *independent* uniform random variables $\{y_j\}_{j \in K}$ that we denote as points $y = (y_1, \ldots, y_k)$ ranging over a *duration space* $D = I_1 \times \cdots \times I_k \subseteq \mathbb{R}^k$ with density $\psi(y_1, \ldots, y_k) = \psi_1(y_1) \cdots \psi_k(y_k)$. A state-based representation of a process is given by simple DPA.

[1] See [1] for a straightforward generalization to partial-order precedence constraints.

Definition 4 (SDPA). *A simple duration probabilistic automaton (SDPA) of k steps is a tuple $\mathcal{A} = (\Sigma, Q, \{x\}, Y, \Delta)$ where $\Sigma = \Sigma_s \uplus \Sigma_e$ is the alphabet of* start *and* end *actions with $\Sigma_s = \{s_1, \ldots, s_k\}$ and $\Sigma_e = \{e_1, \ldots, e_k\}$. The state space is an ordered set $Q = \{\overline{q}_1, q_1, \overline{q}_2, \ldots, q_k, \overline{q}_{k+1}\}$ with $\overline{q}_j$ states considered* idle *and q_j states are* active*, x is a clock variable and $Y = \{y^1, \ldots, y^k\}$ is a set of auxiliary random variables. The transition relation Δ consists of tuples of the form (q, g, r, q') with q and q' being the source and target of the transition, g is a guard, a precondition (external or internal) for the transition and r is an action (internal or external) accompanying the transition. The transitions are of two types:*

1. Start *transitions: for every idle state $\overline{q}_j$, $j < k+1$ there is one transition of the form $(\overline{q}_j, s_j, \{x\}, q_j)$. The transition, triggered by a scheduler command s_j, activates clock x and sets it to zero;*
2. End *transitions: for every active state q_j, there is a transition, conditioned by the clock value, of the form $(q_j, x = y_j, e_j, \overline{q}_{j+1})$. This transition renders clock x inactive and outputs an e_j event.*

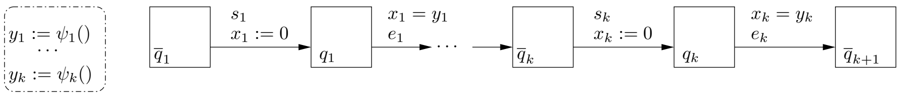


Fig. 1. A simple DPA

For each step j we draw a duration y_j according to ψ_j. Upon a scheduler command s_j the automaton moves from a waiting state $\overline{q}_j$ to active state q_j in which clock x advances with derivative 1. The *end* transition is taken when $x = y_j$, that is, y_j time after the corresponding *start* transition. An extended state of the automaton is a pair (q, x) consisting of a discrete state and a clock value which represents the time elapsed since the last *start* transition. The extended state-space of the SDPA is thus

$$S = \{(\overline{q}_j, \bot) : j \leq k+1\} \cup \{(q_j, x) : j \leq k \wedge x \leq b_j\}$$

where $\bot$ indicates the inactivity of the clock in waiting/idle states.

Note the difference between transition labels s_j and e_j: the *start* transitions are *controllable* and are issued by the scheduler that we want to optimally synthesize while the *end* transitions are generated by the *uncontrolled* external (to the scheduler) environment represented by random variable y_j. Without a scheduler, the SDPA is *underdetermined* and can issue a *start* transition any time. The derivation of an optimal scheduler is done via the computation of a *time-to-go* function that we first illustrate on the degenerate case of one process in isolation. In this case each state has only one successor and any waiting between steps unnecessarily increases the time to termination.

Definition 5 (Local Stochastic Time to Go). *The local stochastic time-to-go function associates with every state (q, x) a time density $\mu(q, x)$ with $\mu(q, x)[t]$ indicating the* *probability to terminate within t time* given *that we start from (q, x) and apply the optimal strategy.*

This function admits the following inductive definition:

$$\mu(\overline{q}_{k+1}, \perp) = \mathbf{0} \tag{1}$$

$$\mu(\overline{q}_j, \perp) = \mu(q_j, 0) \tag{2}$$

$$\mu(q_j, x)[t] = \int_0^t \psi_{j\,/x}[t'] \cdot \mu(\overline{q}_{j+1}, 0)[t - t']dt' \tag{3}$$

Line (1) indicates the final state while (2) comes from the fact that in the absence of conflicts the optimal scheduler need not wait and should start each step immediately when enabled. Equation (3) computes the probability for termination at t based on the probabilities of terminating the current step in some t' and of the remaining time-to-go being $t - t'$. It can be be summarized in a functional language as

$$\mu(q_j, x) = \psi_{j\,/x} * \mu(\overline{q}_{j+1}, \perp) = \psi_{j\,/x} * \mu(q_{j+1}, 0) \tag{4}$$

The successive application of (4) yields, not surprisingly, $\mu(q_1, 0) = \psi_1 * \cdots * \psi_k$.

Definition 6 (Local Expected Time to Go). *The expected time-to-go function is $V : Q \times X \to \mathbb{R}_+$ defined as*

$$V(q, x) = \int \mu(q, x)[t] \cdot t dt = \mathbb{E}(\mu(q, x)).$$

This measure satisfies $V(q_j, x) = \mathbb{E}(\psi_{j\,/x}) + V(q_{j+1}, 0)$ where the first term is the expected termination time of step j starting from x. For the initial state this yields

$$V(q_1, 0) = \mathbb{E}(\psi_1 * \cdots * \psi_k) = \mathbb{E}(\psi_1) + \cdots + \mathbb{E}(\psi_k) = \sum_{j=1}^{k}(a_j + b_j)/2.$$

4 Conflicts and Schedulers

We extend the model to n processes, indexed by $N = \{1..n\}$, that may run in parallel except for steps which are mutually conflicting due to the use of the same resource. For simplicity of notation we assume all processes to have the same number k of steps.

Definition 7 (Process System). *A process system is a triple $(\mathcal{P}, M, h)$ where*

$$\mathcal{P} = P^{\mathbf{1}}|| \cdots ||P^{\mathbf{n}} = \{(\mathcal{I}^{\mathbf{i}}, \Psi^{\mathbf{i}})\}_{i \in N}$$

is a set of processes, M is a set of resources, and $h : N \times K \to M$ is a function which assigns to each step the resource that it uses.

We use notations $P^{\mathbf{i}}_j$ to refer to step j of process i and $\psi^{\mathbf{i}}_j$ and $I^{\mathbf{i}}_j = [a^{\mathbf{i}}_j, b^{\mathbf{i}}_j]$ for the respective densities and their support intervals. Likewise we denote the corresponding controllable and uncontrollable actions by $s^{\mathbf{i}}_j$ and $e^{\mathbf{i}}_j$, respectively. Without loss of generality we assume there is one instance of each resource type, hence any two steps $P^{\mathbf{i}}_j$ and $P^{\mathbf{i}'}_{j'}$ such that $h(i,j) = h(i',j')$ are in *conflict* and cannot execute simultaneously. Each process is modeled as an SDPA $\mathcal{A}^{\mathbf{i}} = (\Sigma^{\mathbf{i}}, Q^{\mathbf{i}}, \{x^{\mathbf{i}}\}, Y^{\mathbf{i}}, \Delta^{\mathbf{i}})$ and the global system is obtained as a product of those restricted to conflict-free states. We write global states as $q = (q^{\mathbf{1}}, \ldots, q^{\mathbf{n}})$ and exclude states where for some i and i', $q^{\mathbf{i}} = q^{\mathbf{i}}_j$, $q^{\mathbf{i}'} = q^{\mathbf{i}'}_{j'}$ and steps $P^{\mathbf{i}}_j$ and $P^{\mathbf{i}'}_{j'}$ are conflicting. We say that action $s^{\mathbf{i}}_j$ (respectively, $e^{\mathbf{i}}_j$) is enabled in q if $q^{\mathbf{i}} = \overline{q}^{\mathbf{i}}_j$ (resp. $q^{\mathbf{i}} = q^{\mathbf{i}}_j$). Since only one transition per process is possible in a global state, we will sometime drop the j-index and refer to those as $s^{\mathbf{i}}$ and $e^{\mathbf{i}}$.

Definition 8 (Duration Probabilistic Automata). *A duration probabilistic automaton (DPA) is a composition $\mathcal{A} = \mathcal{A}^{\mathbf{1}} \circ \cdots \circ \mathcal{A}^{\mathbf{n}} = (\Sigma, Q, X, Y, \Delta)$ of n SDPA with the action alphabet being $\Sigma = \bigcup_i \Sigma^{\mathbf{i}}$. The discrete state space is $Q \subseteq Q^{\mathbf{1}} \times \cdots Q^{\mathbf{n}}$ (with forbidden states excluded). The set of clocks is $X = \{x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}\}$, the extended state-space is $S \subseteq S^{\mathbf{1}} \times \cdots S^{\mathbf{n}}$ and the auxiliary variables are $Y = \bigcup_i Y^{\mathbf{i}}$ ranging over the joint duration space $D = D^{\mathbf{1}} \times \cdots \times D^{\mathbf{n}}$. The transition relation Δ is built using interleaving, that is, a transition (q, g, r, q') from $q = (q^{\mathbf{1}}, \ldots, q^{\mathbf{i}}, \ldots, q^{\mathbf{n}})$ to $q' = (q^{\mathbf{1}}, \ldots, q'^{\mathbf{i}}, \ldots, q^{\mathbf{n}})$ exist in Δ if a transition from $(q^{\mathbf{i}}, g, r, , q'^{\mathbf{i}})$ exists in $\Delta^{\mathbf{i}}$, provided that q' is not forbidden.*

The DPA thus defined (see Fig. 2-a) is not probabilistically correct as it admits non-determinism of a non probabilistic nature: in a given state the automaton may choose between several *start* transitions or decide to wait for an *end* transition (the termination of an active step). A *scheduler* selects *one* action in any state and then the only non-determinism that remains is due to the probabilistic task durations. A discussion on different types of schedulers can be found in [12].

Definition 9 (Scheduler). *A scheduler for a DPA $\mathcal{A}$ is a function $\Omega : S \to \Sigma_s \cup \{\mathbf{w}\}$ such that for every $s \in \Sigma_s$, $\Omega(q, x) = s$ only if s is enabled in q and $\Omega(q, x) = \mathbf{w}$ (wait) only if q admits at least one active component.*

Composing the scheduler with the DPA (see Fig. 2-b) renders it input-deterministic in the sense that any point $y \in D$ induces a unique[2] run of the automaton.

Definition 10 (Steps, Runs and Successors). *The steps of a controlled DPA $\mathcal{A} \circ \Omega$, induced by a point $y \in D$ are of the following types:*

- *Start steps:* $(q, x) \xrightarrow{s^{\mathbf{i}}_j} (q', x')$ *iff* $q^{\mathbf{i}} = \overline{q}^{\mathbf{i}}_j$ *and* $\Omega(q, x) = s^{\mathbf{i}}_j$;
- *End steps:* $(q, x) \xrightarrow{e^{\mathbf{i}}_j} (q', x')$ *iff* $q^{\mathbf{i}} = q^{\mathbf{i}}_j$ *and* $x^{\mathbf{i}} = y^{\mathbf{i}}_j$;
- *Time steps:* $(q, x) \xrightarrow{t} (q, x + t)$ *iff* $\forall i\ (q^{\mathbf{i}} = q^{\mathbf{i}}_j \Rightarrow x^{\mathbf{i}} + t < y^{\mathbf{i}}_j)$.

[2] We define a priority order among the $e^{\mathbf{i}}$-actions so that in the (measure zero) situation where two actions are taken simultaneously we impose the order to guarantee a unique run and avoid artifacts introduced by the interleaving semantics.

The run *associated with y is a sequence of steps starting at $(\overline{q}^{\mathbf{1}}_1, \ldots, \overline{q}^{\mathbf{n}}_1)$ and ending in $(\overline{q}^{\mathbf{1}}_{k+1}, \ldots, \overline{q}^{\mathbf{n}}_{k+1})$.*
The t-i-successor *of a state (q, x) denoted by $\sigma^{\mathbf{i}}(t, q, x)$, is the state (q', x') reached from (q, x) after a time step of duration t followed by a a transition of $P^{\mathbf{i}}$.*

The duration of a run is the sum of the time steps and it coincides with the termination time of the last process, known as *makespan* in the scheduling literature.

5 Expected Time Optimal Schedulers

In [12] we developed a method to compute the expected termination time under a *given* scheduler by computing volumes in the duration space. We now show how to optimally synthesize such schedulers from an uncontrolled DPA description. To this end we extend the formulation of stochastic time-to-go from a single process (3) to multiple processes (7).

We define a partial order relation on global states based on the order on the local states with $q \preceq q'$ if for every i, $q^{\mathbf{i}} \preceq q^{\mathbf{i}'}$. For extended states we let $(q, x) \preceq (q', x')$ if either $q \prec q'$ or $(q = q' \wedge x \leq x')$. The *forward cone* of a state q is the set of all states q' such that $q \prec q'$. The immediate successors of a state q are the states reachable from it by one transition. A *partial scheduler* is a scheduler defined only on a subset of Q. To optimize the decision of the scheduler in a state we need to compare the effect of the possible actions on the time-to-go.

Definition 11 (Local Stochastic Time-to-Go). *Let $\mathcal{A}$ be a DPA with a partial strategy whose domain includes the forward cone of a state q. With every i, x and every $s \in \Sigma_s \cup \{\mathbf{w}\}$ enabled in q, the time density $\mu^{\mathbf{i}}(q, x, s) : \mathbb{R}_+ \rightarrow [0, 1]$ characterizes the stochastic time-to-go for process $P^{\mathbf{i}}$ if the controller issues action s at state (q, x) and continues from there according to the partial strategy.*

Note that for any successor q' of q the optimal action has already been selected and we denote its associated time-to-go by $\mu^{\mathbf{i}}(q', x')$. Once $\mu^{\mathbf{i}}(q, x, s)$ has been computed for every i, the following measures, all associated with action s, can be derived from it.

Definition 12 (Global Stochastic Time-to-Go). *With every state (q, x) and action s enabled in it, we define*

- *The stochastic time-to-go for total termination (makespan) which is the expected duration of the last task: $\mu(q, x, s) = \max\{\mu^{\mathbf{1}}(q, x, s), \ldots, \mu^{\mathbf{n}}(q, x, s)\}$;*
- *The expected total termination time: $V(q, x, s) = \displaystyle\int t \cdot \mu(q, x, s)[t]dt$*

The computation of μ for a state q, based on the stochastic time-to-go of its successors, is one of the major contributions of the paper. The hard part is the computation of the time-to-go associated with *waiting* in a state where several processes are active. In this *race* situation the automaton may leave q via different transitions and μ should be computed based on the probabilities of these transitions (and their timing) and the time-to-go from the respective successor states.

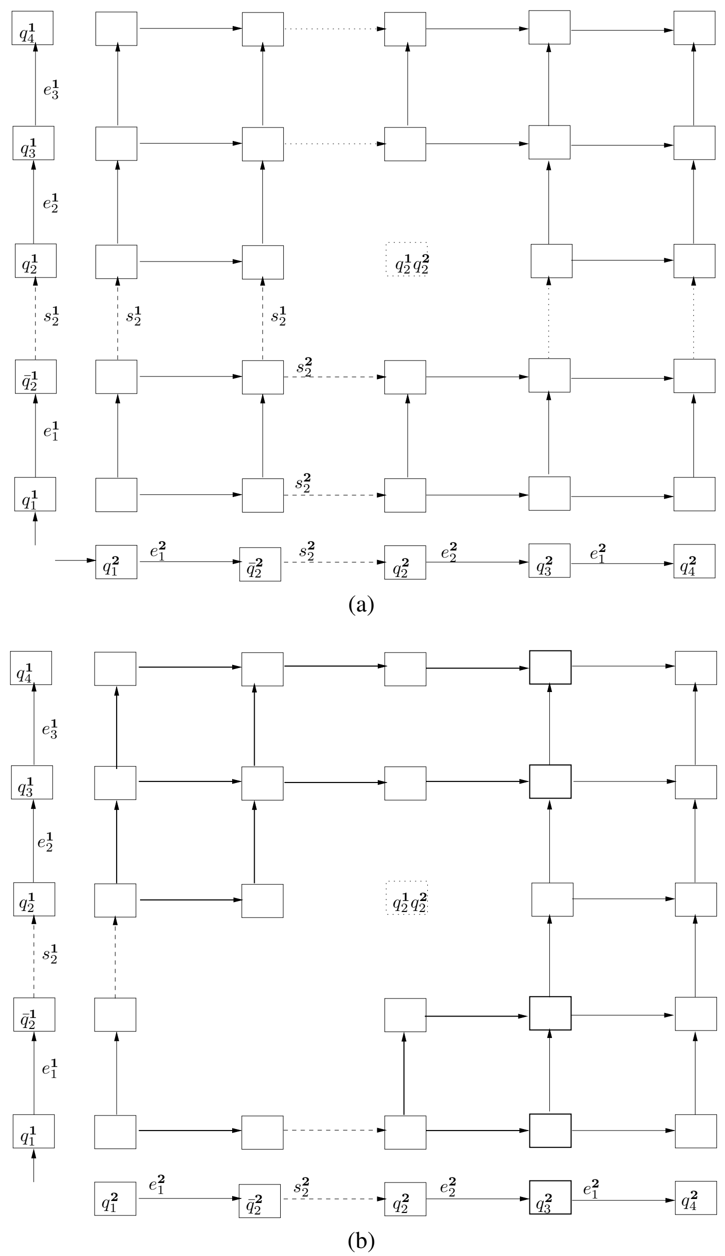


Fig. 2. (a) The DPA for two parallel processes admitting a resource conflict and their respective second steps, with the conflict state (q^1_2, q^2_2) removed. The dashed arrows indicate *start* transitions which should be under the control of a scheduler while the dotted arrows indicate post-conflict *start* transitions; (b) The automaton resulting from composition with a FIFO scheduler which starts a step as soon as it is enabled.

With each state (q, x) we associate a family $\{\rho^{\mathbf{i}}(q, x)\}_{i \in N}$ of partial time densities with the intended meaning that $\rho^{\mathbf{i}}(q, x)[t]$ is (the density of) the probability that the *first* process to terminate its current step is $P^{\mathbf{i}}$ *and* that this occurs within t time. This is relative to the fact that the time elapsed since the initiation of each active step is captured by the respective clock value in x.[3]

Definition 13 (Race Winner). *Let q be a state where n processes are active, each in a step admitting a time density $\psi^{\mathbf{i}}$. With every clock valuation $x = (x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \leq (b^{\mathbf{1}}, \ldots, b^{\mathbf{n}})$ and every i we associate the partial density:*

$$\rho^{\mathbf{i}}(q, x)[t] = \psi^{\mathbf{i}}{}_{/x^{\mathbf{i}}}[t] \cdot \prod_{i' \neq i} \psi^{\mathbf{i}'}{}_{/x^{\mathbf{i}'}}[> t]$$

This is the probability that $P^{\mathbf{i}}$ terminates in t time and every other process $P^{\mathbf{i}'}$ terminates within some $t' > t$.

Definition 14 (Computing Stochastic Time-to-go). *For every i, the function $\mu^{\mathbf{i}}$ is defined inductively as*

$$\mu^{\mathbf{i}}((\ldots \overline{q}^{\mathbf{i}}_{k+1} \ldots), x) = \mathbf{0} \tag{5}$$

$$\mu^{\mathbf{i}}(q, x, s^{\mathbf{i}'}) = \mu^{\mathbf{i}}(\sigma^{\mathbf{i}'}(0, q, x))) \tag{6}$$

$$\mu^{\mathbf{i}}(q, x, \mathbf{w})[t] = \sum_{i'=1}^{n} \int_0^t \rho^{\mathbf{i}'}(q, x)[t'] \cdot \mu^{\mathbf{i}}(\sigma^{\mathbf{i}'}(t', q, x))[t - t']dt' \tag{7}$$

For any global state where $P^{\mathbf{i}}$ is in its final state, $\mu^{\mathbf{i}}$ is zero (5). Each enabled *start* action $s^{\mathbf{i}'}$ leads immediately to the successor state and the cost-to-go is inherited from there (6). For waiting we make a convolution between the probability of $P^{\mathbf{i}'}$ winning the race and the value of $\mu^{\mathbf{i}}$ in the post-transition state and sum up over all the active processes (7). The basic iterative step in computing the value function and strategy is summarized in Algorithm 1. A *dynamic programming* algorithm starting from the final state and applying the above procedure will produce the expected-time optimal strategy. Since we are dealing with *acyclic* systems, the question of convergence to a fixed point is not raised and the only challenge is to show that the defined operators are computable.

6 Computing Optimal Strategies

Algorithm 1 splits into three parts, the third being merely book-keeping. In the first we essentially *compute* the outcome of *waiting* (by race analysis) and of *starting*. As we

[3] Note that the dependence on the time already elapsed is in sharp contrast with the *memoryless exponential* distribution where this time does *not* matter for the future. For those distributions the time-to-go is associated only with the discrete state and is much easier to compute, see [1] for the derivation of optimal schedulers for timed automata with exponential durations.

Algorithm 1: Value Iteration

```
Input: A global state $q$ such that $\Omega(q', x)$ and $\mu^{\mathbf{i}}(q', x)$ have been
  computed for each of its successors $q'$ and every $i$
Output: $\Omega(q, x)$, and $\mu^{\mathbf{i}}(q, x)$

      % COMPUTE:
forall $s \in \Sigma_s \cup \{\mathbf{w}\}$ enabled in $q$
  for $i = 1..n$
    compute $\mu^{\mathbf{i}}(q, x, s)$ according to (6-7)
  end
  compute $\mu(q, x, s)$    (max of random variables)
  compute $V(q, x, s)$    (expected makespan)
end
      % OPTIMIZE:
forall $x \in Z_q$
  $V(q, x) = \min_s(V(q, x, s))$
  $s_* = \arg\min_s V(q, x, s)$
  $\Omega(q, x) = s_*$
end
      % UPDATE:
for $i = 1..n$
  $\mu^{\mathbf{i}}(q, x) = \mu^{\mathbf{i}}(q, x, s_*)$
end
```

shall see, starting from a specific class of time densities, this computation can be done in a symbolic/analytic way, resulting in *closed-form expressions* over x and t for the values of $\mu^{\mathbf{i}}$, μ and V associated with each action. The second part involves *optimization*: to classify extended states according to the action that optimizes V in them. For this part we prove a monotonicity property which facilitates the task of approximating the boundaries between the optimality domains of the various actions.

Each of the functions we have defined is in fact an infinite family of functions parameterized by a state q and clock valuation x ranging over the rectangle Z_q. They can be characterized as follows.

Definition 15 (Zone-Polynomial Time Densities). *A function $\mu : Z \to (\mathbb{R}_+ \to [0, 1])$ over a rectangular clock space Z is zone-polynomial if it can be written as*

$$\mu(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t] = \begin{cases} f_1(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t]) \text{ if } Z_1(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \text{ and } l_1 \le t \le u_1 \\ f_2(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t]) \text{ if } Z_2(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \text{ and } l_2 \le t \le u_2 \\ \ldots \\ f_L(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t]) \text{ if } Z_L(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \text{ and } l_L \le t \le u_L \end{cases}$$

where

- *For every r, $Z_r(x^{\mathbf{1}}, ..., x^{\mathbf{n}})$ is a zone included in the rectangle Z, which moreover satisfies either $Z_r \subseteq [x^{\mathbf{i}} \le a^{\mathbf{i}}]$ or $Z_r \subseteq [a^{\mathbf{i}} \le x^{\mathbf{i}}]$, for every $i = 1..n$.*

- *For every r, the bounds l_r, u_r of the t interval are either nonnegative integers c or terms of the form $c - x^{\mathbf{i}}$, with $i = 1..n$, $c \in \mathbb{Z}^+$. Moreover, the interval $[l_r, u_r]$ must be consistent with the underlying zone, that is, $Z_r \subseteq [l_r, u_r]$.*
- *For every r, $f_r(x^{\mathbf{1}}, ..., x^{\mathbf{n}})[t] = \sum\limits_k \frac{P_k(x^{\mathbf{1}},...,x^{\mathbf{n}})}{Q_r(x^{\mathbf{1}},...,x^{\mathbf{n}})} t^k$ where P_k are arbitrary polynomials and Q_r is a characteristic polynomial associated with zone Z_r defined as $\prod\limits_i (b^{\mathbf{i}} - \max\{x^{\mathbf{i}}, a^{\mathbf{i}}\})$. Note that for each zone, the* max *is attained uniformly as either $a^{\mathbf{i}}$ or $x^{\mathbf{i}}$.*

Theorem 1 (Closure of Zone-Polynomial Densities). *Zone-polynomial time densities are closed under (6) and (7).*

Sketch of Proof: Operation 6 is a simple substitution. Closure under summation is also evident - you just need to refine the partitions associated with the summed functions and make them compatible and then apply the operation in each partition block. The only intricate operation is the quasi-convolution part of (7). The function $\mu^{\mathbf{i}}(\sigma^{\mathbf{i}'}(t', q, x))[t - t']$ is *not* a zone polynomial time density. Due to the time progress by t' enforced by the substitution $\sigma^{\mathbf{i}'}$ it might happen that polynomials of the form $(b_i - (x^{\mathbf{i}} - t'))$ appear in the denominators. But, in all feasible cases, they will be simplified through multiplication by $\rho^{\mathbf{i}'}(q, x)[t']$ which contains the same polynomials as factors (within $\psi^{\mathbf{i}}{}_{/x^{\mathbf{i}}}[\geq t']$, see Def. 13). Hence, integration of t' is always trivially completed as t' occurs only on numerator polynomials and/or powers of the form t'^k and $(t - t')^k$. Moreover, after integration, the remaining constraints on t and x can also be rewritten to match the required form of zone-polynomial time densities. ▟

Next we move to the *optimization* part of the iteration. Consider a state q where process $P^{\mathbf{i}}$ has to decide whether or not to start a step while other processes are active in their respective steps. The optimal strategy Ω in this state partitions the clock space of the other processes into $\Omega^{-1}(s^{\mathbf{i}})$ and $\Omega^{-1}(\mathbf{w})$, extended states where waiting or starting are preferred. The boundary corresponds to the set of solutions of the piecewise-polynomial equation $V(q, x, s) = V(q, x, \mathbf{w})$. Our approach is to *approximate* the optimal strategy based on sampling: we cut the clock space Z_q into an ε-grid and for each grid point x we compare $V(q, x, s)$ and $V(q, x, \mathbf{w})$ and select the optimal value (Fig. 3-(a)). Once the optimal action has been computed for all grid points we complete the strategy for the rest of the clock-space by selecting in each ε-cube the action which is optimal for its leftmost corner (Fig. 3-(b)). To estimate the deviation from the optimal strategy we first bound the derivative of V with respect to any of the clocks.

Lemma 1 (Derivative of V). *Let V be the value function associated with a problem. Then for every (q, x) and for every i* $\dfrac{\partial V}{\partial x^{\mathbf{i}}}(q, x) \geq -1$

Sketch of Proof: Consider first the local value function of a process in isolation. In any state q, the clock space splits into two parts. When $x < a$, the derivative is naturally -1. When $x > a$ the rate of progress is slower (because progress is combined with accumulation of optimism-contradicting information) and the magnitude of the

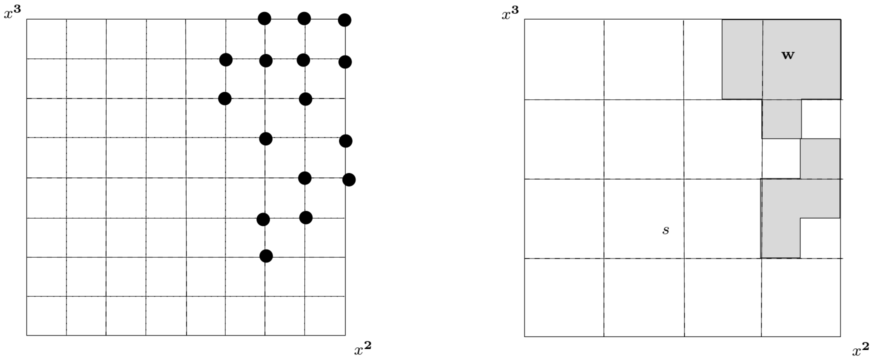


Fig. 3. Approximating the optimal action for P^1 in the clock space of $P^2||P^3$: (a) Computing the optimal action for all grid points with the dark circle indicating $\Omega^{-1}(\mathbf{w})$; (b) Approximating the optimal strategy with the dark cubes indicating $\Omega'^{-1}(\mathbf{w})$.

derivative is smaller.[4] When running together, the progress of each process is bounded by its progress in isolation or by the progress of another process that blocks it. The progress of the expected termination of the last process (makespan) is bounded by the progress of each of the processes. ▟

Lemma 2 (Approximation). *Let x' be a point in the clock space and let $x < x'$ be its nearest grid point on the left and hence $d(x, x') < \varepsilon$. Assume without loss of generality that the optimal strategy Ω satisfies $\Omega(q, x) = s$ and $\Omega(q, x') = \mathbf{w}$. Consider an approximate strategy Ω' such that $\Omega'(q, x') = s$, then the respective value functions of the strategies satisfy $V'(q, x') - V(q, x') \leq \varepsilon$.*

Proof: According to what we know about the optimal strategy we have

$$V(q, x', \mathbf{w}) < V(q, x', s) < V(q, x, s) < V(q, x, \mathbf{w})$$

and from Lemma 1, $V(q, x, s) - V(q, x', \mathbf{w}) \leq \varepsilon$ and so is $V(q, x', s) - V(q, x', \mathbf{w}) = V'(q, x') - V(q, x')$. ▟

Before we state the consequent main result, we prove an important property of optimal strategies which can reduce the number of grid points for which the optimal strategy should be computed. This "non-laziness" property, formulated in the deterministic setting in [1], simply captures the intuition that preventing an enabled process from taking a resource is *not* useful *unless* some other process benefits from the resource during the waiting period.[5] The proof in a non-deterministic setting is more involved.

[4] The derivative of V represents the progress toward the average duration, minus the growth of the average itself when $x > a$.

[5] Or, in scheduling under uncertainty, if some information gathered during the period has increased the expected time-to-go associated with waiting, which is impossible in our setting. Non-lazy schedules are also known as *active* schedules.

Definition 16 (Laziness). *A scheduling policy Ω is lazy at (q,x) if $\Omega(q,x+t) = s^{\mathbf{i}}$ for some i and $\Omega(q,x+t') = \mathbf{w}$ for every $t' \in [0,t)$. A schedule is non-lazy if no such (q,x) exists.*[6]

Theorem 2 (Non Lazy Optimal Schedulers). *The optimal value V can be obtained by a non-lazy scheduler.*

To prove the theorem we need a lemma whose proof is omitted for lack of space.

Lemma 3 (Value of Progress). *Let q be a state and let x and x' be two clock valuations which are identical except for $x'^{\mathbf{i}} = x^{\mathbf{i}} + \delta$. Then the value of (q,x') is at least as good as the value of (q,x), that is, $V(q,x') \leq V(q,x)$.*

The lemma can be wrongly interpreted as saying that always a more advanced state has a better value. This is true in general only for progress that does *not* induce a possibility of *blocking* other processes: advancing the clock of an already-active step or starting a step on a resource that has no other future users. Starting a step on a resource that has other users in its horizon is a type of progress which is outside the scope of the lemma.

Proof of Theorem 2: Imagine a strategy Ω which is lazy at (q,x) and takes its earliest *start* at $(q,x+t)$, as illustrated in Fig. 4-(a). Following an alternative strategy that starts at x, we will find ourselves after t time in a state where one clock is more advanced (Fig. 4-(b)) and hence satisfying the condition of Lemma 3. If we keep all instances of laziness partially ordered according to $\preceq$ and apply the above modification starting from the minimal elements of the state space, we gradually push the earliest laziness toward later states until it is completely eliminated. ∎

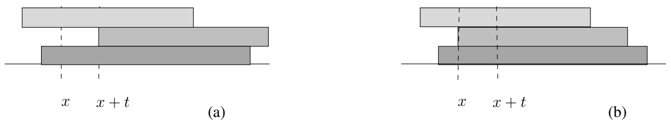


Fig. 4. Proof of theorem: the state reached after starting at $x+t$ (a) is less advanced than after starting at x (b)

To illustrate the limited scope of the lemma and theorem consider a task whose duration is characterized by a *discrete* probability with probability p for a and $1-p$ for b. In this case, the value function associated with waiting is

$$V(x) = \begin{cases} p(a-x) + (1-p)(b-x) & \text{when } x < a \\ 0(a-x) + 1(b-x) & \text{when } x > a \end{cases}$$

Here at $x = a$ there is a *jump* in V from $(1-p)(b-a)$ to $(b-a)$ which is, intuitively, due to the accumulation of information: after a time, the non-occurrence of an *end* event

[6] This definition of laziness can be extended from a one-dimensional interval $[x, x+t]$ to any pair of clock valuations x and x' such that x' is reachable from x by a time step.

tells us that the duration is certainly b. Such a situation contradicts the lemma, because for $x < a < x'$ we may have $V(x') > V(x)$. This jump in the expected time-to-go for waiting can also justify laziness: when $x < a$ the expected value of waiting can be better than for starting, but after $x = a$ the relation between these values may change.

Corollary 1 (Forward-Backward Closure). *Let (q, x) be an extended state such that $V(q, x, \mathbf{w}) < V(q, x, s)$ and hence the optimal scheduler satisfies $\Omega(q, x) = \mathbf{w}$. Then $\Omega(q, x') = \mathbf{w}$ for any $x' = x + t$. Likewise if $V(q, x, \mathbf{w}) > V(q, x, s)$ then $\Omega(q, x') = s$ for all $x' = x - t$.*

This property can be used to reduce the number of grid points for which the strategy should be computed: $\Omega(q, x) = \mathbf{w}$ implies $\Omega(q, x') = \mathbf{w}$ for any grid point of the form $x' = x + m\varepsilon$ and likewise if $\Omega(q, x) = s$, starting is also optimal for any $x' = x - m\varepsilon$. Using an adaptation of the multi-dimensional binary search algorithm of [16], originally devised to approximate Pareto fronts for multi-criteria optimization problems, one can complete the marking of grid points with less than $(1/\varepsilon)^n$ evaluations of the strategies.

Theorem 3 (Main Result). *Let Ω be the expected-time optimal scheduler whose value at the initial state is V. For any ε, one can compute a scheduler Ω' whose value V' satisfies $V' - V \leq \varepsilon$.*

Proof: Apply Algorithm 1 while going backwards from the final state. In the optimization part use sampling with grid size ε/nk. The division by nk is needed to tackle the (theoretical) possibility of approximation error accumulation along paths. ∎

7 Discussion

To the best of our knowledge, this work presents the first automatic derivation of optimal controllers/schedulers for "non-Markovian" continuous-time processes. We have laid down the conceptual and technical foundation for treating *clock-dependent* time densities and value functions and investigated the properties of optimal schedulers. We list below some ongoing and future work directions:

1. The algorithm as described here works individually on each global state, which is a recipe for quick state explosion. A more sophisticated algorithm that works on the whole decision subspace associated with an action s and taking advantage of forward/backward closure, will be much more efficient. Although the stronger property of upward-downward closure does not follow, unfortunately, from non-laziness, the multi-dimensional binary search algorithm of [16] could provide good approximations whose error analysis will require a more refined characterization of the partition induced by the optimal strategy.
2. Implementation and experimentation: we currently have a prototype implementation building upon the infra-structure developed in [12] for computing integrals over zones. Once completed, we intend to compare cost/quality tradeoffs of our algorithm with statistical methods such as [8,13] that evaluate and synthesize schedulers based on sampling the duration space. Such experiments will determine whether symbolic techniques can be part of tools for design-space exploration [13].

3. Extending the result to cyclic systems will require a definition of the value associated with infinite runs and some new techniques for proving convergence to a fixed point in the spirit of [3]. Introducing stochasticity in task arrival will enrich queuing theory with the expressive advantage of automata.
4. One can think of alternative measures for evaluating the performance of the scheduler. For example, rather than considering the expected makespan ($\max$ over all processes) we can optimize the average termination time of all tasks, or even employ a multi-dimensional vector of termination times in the multi-criteria spirit.

Acknowledgement. The project benefitted from the support of the French ANR project EQINOCS and numerous useful comments given by E. Asarin.

References

1. Abdeddaïm, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theoretical Computer Science 354(2), 272–300 (2006)
2. Alur, R., Bernadsky, M.: Bounded Model Checking for GSMP Models of Stochastic Real-Time Systems. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 19–33. Springer, Heidelberg (2006)
3. Asarin, E., Degorre, A.: Volume and Entropy of Regular Timed Languages: Analytic Approach. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 13–27. Springer, Heidelberg (2009)
4. Asarin, E., Maler, O.: As Soon as Possible: Time Optimal Control for Timed Automata. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 19–30. Springer, Heidelberg (1999)
5. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: Proc. IFAC Symposium on System Structure and Control, pp. 469–474 (1998)
6. Carnevali, L., Grassi, L., Vicario, E.: State-density functions over DBM domains in the analysis of non-Markovian models. IEEE Trans. Software Eng. 35(2), 178–194 (2009)
7. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer (2008)
8. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
9. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The Tool KRONOS. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 208–219. Springer, Heidelberg (1996)
10. German, R.: Non-Markovian Analysis. In: Brinksma, E., Hermanns, H., Katoen, J.-P. (eds.) FMPA 2000. LNCS, vol. 2090, pp. 156–182. Springer, Heidelberg (2001)
11. Glynn, P.W.: A GSMP formalism for discrete event systems. Proceedings of the IEEE 77(1), 14–23 (1989)
12. Kempf, J.-F., Bozga, M., Maler, O.: Performance Evaluation of Schedulers in a Probabilistic Setting. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 1–17. Springer, Heidelberg (2011)
13. Kempf, J.-F.: On Computer-Aided Design-Space Exploration for Multi-Cores. PhD thesis, University of Grenoble (October 2012)
14. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg (2001)

15. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)
16. Legriel, J., Le Guernic, C., Cotton, S., Maler, O.: Approximating the Pareto Front of Multi-criteria Optimization Problems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 69–83. Springer, Heidelberg (2010)
17. Maler, O.: On optimal and reasonable control in the presence of adversaries. Annual Reviews in Control 31(1), 1–15 (2007)
18. Maler, O., Larsen, K.G., Krogh, B.H.: On zone-based analysis of duration probabilistic automata. In: INFINITY, pp. 33–46 (2010)

Integer Parameter Synthesis for Timed Automata*

Aleksandra Jovanović, Didier Lime, and Olivier H. Roux

LUNAM Université. École Centrale de Nantes - IRCCyN UMR CNRS 6597
Nantes, France

Abstract. We provide a subclass of parametric timed automata (PTA) that we can actually and efficiently analyze, and we argue that it retains most of the practical usefulness of PTA. The currently most useful known subclass of PTA, L/U automata, has a strong syntactical restriction for practical purposes, and we show that the associated theoretical results are mixed. We therefore advocate for a different restriction scheme: since in classical timed automata, real-valued clocks are always compared to integers for all practical purposes, we also search for parameter values as bounded integers. We show that the problem of the existence of parameter values such that some TCTL property is satisfied is PSPACE-complete. In such a setting, we can also of course synthesize all the values of parameters and we give symbolic algorithms, for reachability and unavoidability properties, to do it efficiently, i.e., without an explicit enumeration. This also has the practical advantage of giving the result as symbolic constraints between the parameters. We finally report on a few experimental results to illustrate the practical usefulness of the approach.

1 Introduction

Real-time systems are ubiquitous, and to ensure their correct design it seems natural to rely on the mathematical framework provided by formal methods. Within that framework, the model-checking of timed models is becoming ever more efficient. It nevertheless requires a complete knowledge of the system. Consequently, the verification can only be performed after the design stage, when the global system and its environment are known. Getting a complete knowledge of a system is often impossible and even when it is possible, it increases the complexity of the conception and the verification of systems. Moreover, if the model of the system is proved wrong or if the environment changes, this complex verification process must be carried out again. It follows that the use of parametric timed models is certainly a very interesting approach for the design of real-time systems.

However, for general parametric formalisms such as Parametric Timed Automata, the existence of a parameter value such that some state is reachable is

* This work was partially funded by the ANR national research program ImpRo (ANR-2010-BLAN-0317).

undecidable and there is currently no algorithm that solves the synthesis problem of parameter values except for severely restricted subclasses, whose practical usability is unclear.

It is then a challenging issue to define a subclass of parametric timed automata, which retains enough of its expressive power and such that, for both reachability and unavoidability properties, the existence of parameter values is decidable and for which there exist efficient symbolic synthesis algorithms.

1.1 Related Work

Parametric timed automata (PTA) have been introduced by Alur et al. in [3], as a way to specify parametric timing constraints. They study the parametric emptiness problem which asks if there exists a parameter valuation such that the automaton has an accepting run. The problem is proven undecidable for PTA that uses three clocks and six parameters, and applies to both dense and discrete time domains. In [9], the undecidability proof is extended for parametric timed automata that use only strict inequalities. Further in [12], Hune et al. identify a class of parametric timed automata, lower bound/upper bound (L/U) automata, for which the emptiness problem is decidable. However, their model-checking algorithm, that uses Difference Bound Matrix as data structure, might not terminate. Decidability results for L/U automata have been further investigated by Bozzelli and La Torre in [6]. They consider infinite accepting runs and liveness property, and show that main decision problems such as emptiness, finiteness and universality for the set of parameter valuations are decidable and PSPACE-complete. They also study constrained version of emptiness and universality, where parameters are constrained by linear system of equalities and inequalities, and obtain decidability if parameters of different types are not compared in the linear constraint. They show how to compute the explicit representation of the set of parameters, when all the parameters are of the same type (L-automata and U-automata).

An approach for the verification of Parametric TCTL (PTCTL) formulae has been developed in [21] by Wang, where the problem has been proved decidable. A more general problem is studied in [7], where parameters are allowed both in the model and desired property (PTCTL formula). The authors show that the model-checking problem is decidable and the parameter synthesis problem is solvable, in discrete time, over a PTA with one parametric clock, if equality is not allowed in the formulae.

In [4], the authors develop a synthesis algorithm that starts from a reference parameter valuation and derives constraints on parameters, ensuring that the behaviors of PTA are time-abstract equivalent. They also give a conjecture for the termination being true on the examples studied. Henzinger et al. in [10], study more general, hybrid, systems extended with parameters. Their state-space exploration algorithms have been implemented in the model-checking tool HyTech. In [20], the authors analyze time Petri nets with parameters in timing constraints. A property is given as a PTCTL formula, but their model-checking algorithm consists in analysis of a region graph for each parameter valuation.

In [19], the authors extend time Petri nets with inhibitor arcs with parameters, and propose an abstraction of the parametric state-space and semi-algorithms for the parametric synthesis problem, considering simple PTCTL formulae.

1.2 Contributions

L/U-automata can be seen as the most useful subclass of PTA supported by many decidability results for reachability-like properties. We show that, even for the subclass with only upper bounds (U-automata), the existence of parameter valuations such that some unavoidability property is satisfied is undecidable though. We also pinpoint some difficulties with the actual synthesis of parameter values for L/U automata and reachability properties.

We therefore propose a different way of subclassing PTA: instead of syntactical restrictions of guards and invariants we propose a novel approach based on restricting the possible values of the parameters. To obtain decidability results, we show that we have to restrict these values to bounded integers. From a practical point of view, the subclass of PTA in such setting is not that restrictive since the temporal constraints for time automata are usually defined on natural (or rational) numbers. Nevertheless, this subclass is restrictive enough to make the problems we address decidable and to allow symbolic synthesis algorithms of parameter values.

We give symbolic algorithms to synthesize the set of all parameters valuations for reachability and unavoidability properties, without having to enumerate all the possibilities. These algorithms are implemented in our tool, Roméo.

Finally, we show that the problem of the existence of bounded integer valuations for PTA such that some property is satisfied is PSPACE-complete for a significant number of properties, which include Timed Computation Tree Logic (TCTL), and also that lifting either of the boundedness or the integer assumption leads to undecidability even for reachability.

1.3 Organization of the Paper

Section 2 gives the basic definitions related to the formalism of parametric timed automata. Section 3 recalls the main positive results on L/U-automata and gives new negative results that make more precise the practical usefulness of that model. This motivates a different restriction scheme based on limiting the possible values of the parameters. Section 4 presents symbolic algorithms for the synthesis problems when valuations are searched as bounded integers. In its development this section also exhibits semi-algorithms for the general setting and the (unbounded) integer setting. Section 5 gives the computational complexities of the associated problems. Finally, section 6 discusses the performance in practice of the proposed approach, illustrated on a small but realistic case-study.

2 Parametric Timed Automata

$\mathbb{Z}$ is the set of integers and $\mathbb{N}$ the set of natural numbers. $\mathbb{R}$ is the set of real numbers. $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers and $\mathbb{R}_{>0} = \mathbb{R}_{\geq 0} \setminus \{0\}$.

For any closed interval $[a, b]$ of $\mathbb{R}$ with $a, b \in \mathbb{Z}$, we denote by $[a..b]$ its intersection with $\mathbb{Z}$.

Let X be a finite set. 2^X denotes the powerset of X and $|X|$ the size of X.

A *linear expression* on X is an expression generated by the following grammar, for $k \in \mathbb{Z}$ and $x \in X$: $\lambda ::= k \mid k * x \mid \lambda + \lambda$

W.l.o.g we consider *reduced* linear expressions λ in which each element of X occurs at most once and with at most one constant term. We let $\mathsf{Coeff}(\lambda, x)$ denote the coefficient of variable $x \in X$ in λ. If x does not occur in λ then $\mathsf{Coeff}(\lambda, x) = 0$. $\mathsf{Coeff}(\lambda, x)$ is well defined since λ is reduced.

$\wedge$ denotes the logical conjunction. A *linear constraint* on x is an expression generated by the following grammar, with λ a linear expression on X, $\sim \in \{>, \geq\}$: $\gamma ::= \lambda \sim 0 \mid \gamma \wedge \gamma$.

Let $V \subseteq \mathbb{R}$. A V-valuation for X is a function from X to V. We denote by V^X the set of V-valuations on X.

For any subset $X' \subseteq X$, and a V-valuation v on X, we define the restriction of v to X' as the unique V-valuation on X' such that $v_{|X'}(x) = v(x)$. If Y is a set of valuations on X, then $Y_{|X'}$ denotes its projection on X',*i.e.*, $Y_{|X'} = \{v_{|X'} \mid v \in Y\}$.

For a linear expression (resp. constraint) λ on X and a V-valuation v of X, we denote by $v(\lambda)$ the real number (resp. boolean value) obtained by replacing in λ each element x of X by the real value $v(x)$. We denote by $\mathcal{C}(X)$ the set of linear constraints on X.

Given some arbitrary order on X, a valuation can be seen as a real vector of size $|X|$. The set of valuations satisfying some linear constraint is then a *convex polyhedron* of $\mathbb{R}^{|X|}$.

Let X (resp. P) be a finite set. We call *clocks* (resp. *parameters*) the elements of X (resp. P). A *simple* (parametric clock) constraint on X (and P) is a linear constraint on $X \cup P$ such that exactly one element x of X occurs in each conjunct of the expression (not necessarily the same for each conjunct), and $\mathsf{Coeff}(x) \in \{-1, 1\}$. We denote by $\mathcal{B}(X, P)$ the set of such simple constraints and $\mathcal{B}'(X, P)$ the set of simple constraints in which the clock variable always has coefficient -1. As before, for any V-valuation v on P, and any simple constraint γ, $v(\gamma)$ is the linear constraint on X obtained by replacing each parameter $p \in P$ by the real value $v(p)$.

We further define the null valuation $\mathbf{0}_X$ on X by $\forall x \in X, \mathbf{0}_X(x) = 0$. For any subset R of X, and any valuation on X, we denote by $v[R]$ the valuation on X such that $v[R](x) = 0$ if $x \in R$ and $v[R](x) = v(x)$ otherwise. Finally $v + d$, for $d \geq 0$, is the valuation such that $(v + d)(x) = v(x) + d$ for all $x \in X$.

Definition 1 (Parametric Timed Automaton). *A* Parametric Timed Automaton *(PTA) $\mathcal{A}$ is a tuple $(L, l_0, \Sigma, X, P, E, \mathsf{Inv})$ where: L is a finite set of* locations*; $l_0 \in L$ is the* initial *location; Σ is a finite set of* actions*; X is a finite set of* clocks*; P is a finite set of* parameters*; $E \subseteq L \times \Sigma \times \mathcal{B}(X, P) \times 2^X \times L$ is a finite set of* edges*: if $(l, a, \gamma, R, l') \in E$ then there is an edge from l to l' with action a, (parametric)* guard *γ and set of clocks to reset R; $\mathsf{Inv} : L \to \mathcal{B}'(X, P)$ assigns a (parametric)* invariant *to each location.*

For any $\mathbb{Q}$-valuation v on P, the structure $v(\mathcal{A})$ obtained from $\mathcal{A}$ by replacing each simple constraint γ by $v(\gamma)$ is a *timed automaton* with invariants [2,11] (TA).

The behavior of a PTA $\mathcal{A}$ is described by that of all the timed automata obtained by considering all possible valuations of the parameters.

Definition 2 (Semantics of a PTA). *Let $\mathcal{A} = (L, l_0, \Sigma, X, P, E, \mathsf{Inv})$ be a PTA and v be a $\mathbb{R}$-valuation on P. The semantics of $v(\mathcal{A})$ is given by the timed transition system $(Q, q_0, \rightarrow)$ with:*

- *$Q = \{(l, u) \in L \times \mathbb{R}_{\geq 0}^{X} \mid u(v(\mathsf{Inv}(l)))$ is true$\}$;*
- *$q_0 = (l_0, \mathbf{0}_X)$ ($q_0 \in Q$ due to the special form of invariants);*
- *Time transitions: $(l, u) \xrightarrow{d} (l, u+d)$, with $d \geq 0$, iff $\forall d' \in [0, d], (l, u+d') \in Q$;*
- *Action transitions: $(l, u) \xrightarrow{a} (l', u')$, with $a \in \Sigma$, iff $(l, u), (l', u') \in Q$, there exists an edge $(l, a, \gamma, R, l') \in E$, $u' = u[R]$ and $u(v(\gamma))$ is true.*

A finite *run* is a finite sequence $\rho = q_1 a_1 q_2 a_2 \ldots a_{n-1} q_n$ such that for all i, $q_i \in Q$, $a_i \in \Sigma \cup \mathbb{R}_{\geq 0}$ and $q_i \xrightarrow{a_i} q_{i+1}$. For any run ρ, we define $\mathsf{Edges}(\rho) = e_1 \ldots e_m$ as the sequence of edges of the automaton taken in the discrete transitions along the run. We suppose without loss of generality that these edges are indeed thus uniquely defined. A run is *maximal* if it either is infinite or cannot be extended. We denote by $\mathsf{Runs}(v(\mathcal{A}))$ the set of runs that start in the initial state of $v(\mathcal{A})$.

We can define several interesting parametric problems on PTA. Among them we can ask: does there exist valuations for the parameters such that some property is satisfied? And, even more interesting, can we compute all of these valuations? Given a class of problems $\mathcal{P}$ (e.g. reachability, unavoidability, TCTL model-checking, control) these two questions translate into what we respectively call the $\mathcal{P}$-emptiness and the $\mathcal{P}$-synthesis problems:

$\mathcal{P}$-emptiness problem
Inputs : A PTA $\mathcal{A}$ and an instance ϕ of $\mathcal{P}$
Problem : Is the set of valuations v of the parameters such that $v(\mathcal{A})$ satisfies ϕ empty?

$\mathcal{P}$-synthesis problem
Inputs : A PTA $\mathcal{A}$ and an instance ϕ of $\mathcal{P}$
Problem : Compute the set of valuations v of the parameters such that $v(\mathcal{A})$ satisfies ϕ.

In this paper we mainly focus on reachability and unavoidability properties and call the corresponding problems EF and AF. Thus, given a PTA $\mathcal{A}$ and a subset G of its locations, EF-emptiness asks: does there exist a valuation v of the parameters such that G is reachable in $v(\mathcal{A})$? And AF-emptiness asks: does there exist a valuation v of the parameters such that all maximal runs in $v(\mathcal{A})$ go through G? The related synthesis problems immediately follow.

In [3], the EF-emptiness problem was proved undecidable for PTA. We give further negative results in the next section.

3 L/U Automata

The following syntactic subclass of PTA, called L/U-automaton, has been proposed in [12] as a decidable subclass for the EF-emptiness problem. It relies on the notion of upper and lower bounds for parameters:

Definition 3 (Lower and upper bounds). *Let γ be a single conjunct of a simple clock constraint on the set of clocks X and the set of parameters P. Let x be the clock variable occurring in γ. γ is an upper (resp. lower) bound constraint if* $\mathsf{Coeff}(x)$ *is negative (resp. positive).*

p is an upper (resp. lower) bound in the PTA $\mathcal{A}$ if for each conjunct γ of each simple clock constraint in the guards and invariants of $\mathcal{A}$, either $\mathsf{Coeff}(\gamma, p) = 0$ *or p is an upper (resp. lower) bound in γ.*

Definition 4 (L/U-automaton). *A PTA $\mathcal{A}$ is a L/U-automaton if every parameter is either an upper bound or a lower bound in $\mathcal{A}$ but not both.*

3.1 Emptiness

EF-emptiness is PSPACE for L/U automata [12] and, more generally, emptiness, universality and finiteness of the valuation set are PSPACE-complete for infinite runs acceptance properties [6]. These good results are based on a *monotonicity* property that L/U automata have: decreasing lower bounds or increasing upper bounds only *add* behaviors. So if we set all lower bounds to 0 and all upper bounds to a large enough constant that we can compute, then the resulting timed automaton contains all the possible behaviors. This makes these automata very well-suited for reachability-like properties. For other properties however this is not enough. For AF properties, increasing lower bounds or decreasing upper bounds can suppress a run that was a counter-example to the property, and then make this property true.

We now indeed prove, with a reduction from the halting problem of 2-counter machines [17], that the AF-emptiness problem for L/U automata is undecidable. We actually prove it in a more general setting by addressing a further subclass of L/U-automata that allows only for upper bounds:

Definition 5 (L- and U-automaton). *A PTA $\mathcal{A}$ is a U-automaton (resp. L-automaton) if every parameter is an upper (resp. lower) bound in $\mathcal{A}$.*

Theorem 1. *The AF-emptiness problem is undecidable for U-automata.*

3.2 Synthesis

In [6] the authors prove that for L-automata and U-automata, the solution to the synthesis problem for infinite runs acceptance properties can be explicitly computed as a linear constraint of size doubly-exponential in the number of parameters. That is to say this solution can be expressed as a finite union of convex polyhedra.

With a different look at the idea used in [6] to prove that the *constrained* (i.e. with initial constraints) emptiness problem for infinite runs acceptance properties is undecidable for L/U-automata, we can express a new and quite strong result on the solution to the EF-synthesis problem for L/U-automata.

Theorem 2. *If it can be computed, the solution to the EF-synthesis problem for L/U-automata cannot be represented using any formalism for which emptiness of the intersection with equality constraints is decidable.*

Note that, in particular, Theorem 2 rules out the possibilty of computing the solution set as a finite union of polyhedra.

4 Integer Parametric Problems

The decidability results related to emptiness problems for L/U-automata are mixed: properties related to reachability are decidable but very simple properties that are not compatible with the monotonicity property, like unavoidability, are undecidable even for the very restricted subclass of U-automata. As for the actual synthesis of the constraints between parameters that describe the set of valuations that satisfy even the simple case of reachability properties, we have to resort to L- or U- automata, that have severely restricted modeling capacities.

We therefore advocate for different kinds of restrictions to PTA. Note that with only one irrational constant in the guards of timed automata, reachability is undecidable [16]. For all practical purposes these constants are actually always chosen as integers. Even if we insist on rationals, we can make those integers through adequate scaling and we usually have to since most tools only allow them as integers. So, instead of using syntactical restrictions in the guards and invariants of PTA, we think it makes a lot of sense to search for parameter values as *bounded integers*.

We therefore focus on synthesizing (or just proving the existence of) integer valuations for the parameters: a valuation v on a set X is an integer valuation if $\forall x \in X, v(x) \in \mathbb{Z}$. This induces new emptiness and synthesis problems that we call *integer* problems (e.g., integer EF-emptiness problem).

By insisting that these integer values should be bounded we will be (unsurprisingly) able to make all parametric problems decidable, provided the associated non-parametric problem, obtained by choosing one particular valuation, is decidable of course.

These decidability results are however only interesting for practical purposes if we can solve the corresponding synthesis problems symbolically, *i.e.*, without explicitly enumerating all the possible valuations.

To this end, we first introduce symbolic semi-algorithms to solve the synthesis problems in the general setting (possibly non integer valuations) that are based on a quite straightforward extension of the symbolic zone-based state-space exploration that is ubiquitous for timed automata [14].

4.1 Symbolic states for PTA

We therefore extend the notion of symbolic state for PTA, as well as the usual operators associated to them:

Definition 6 (Symbolic state). *A* symbolic state *of a PTA $\mathcal{A}$, with set of clocks X and set of parameters P, is a pair (l, Z) where l is a location of $\mathcal{A}$ and Z is a set of valuations on $X \cup P$.*

For state space computation, we define classical operations on valuation sets:

- future: $Z^{\nearrow} = \{v' \mid v'(x) = v(x) + d, d \geq 0 \text{ if } x \in X; v'(x) = v(x) \text{ if } x \in P\}$;
- reset of the clock variables in set $R \subseteq X$: $Z[R] = \{v[R] \mid v \in Z\}$;
- initial symbolic state of the PTA $\mathcal{A} = (L, l_0, \Sigma, X, P, E, \mathsf{Inv})$: $\mathsf{Init}(\mathcal{A}) = (l_0, \{v \in \mathbb{R}^{X \cup P} \mid v_{|X} \in \{\mathbf{0}_X\}^{\nearrow} \cap v_{|P}(\mathsf{Inv}(l_0))_{|X}\})$;
- successor by edge $e = (l, a, \gamma, R, l')$: $\mathsf{Succ}((l, Z), e) = (l', (Z \cap \gamma)[R]^{\nearrow} \cap \mathsf{Inv}(l'))$

It follows from [12] that all reachable symbolic states of a PTA are convex polyhedra. Also, the following properties trivially hold:

Property 1. The symbolic state abstraction satisfies: (1) Succ is non decreasing, and for any reachable symbolic state (l, Z): (2) Z is convex, (3) if v is an integer parameter valuation then $v(Z)$ is a (convex) zone with integer vertices and (4) for any edge e, $\mathsf{Succ}((l, Z), e) = \bigcup_{v \in Z_{|P}} \mathsf{Succ}((l, v(Z)), e)$

We can extend the Succ operator to a sequence of edges $e_1 \dots e_n$ by defining $\mathsf{Succ}((l, Z), e_1 e_2 \dots e_n) = \mathsf{Succ}(\dots \mathsf{Succ}(\mathsf{Succ}((l, Z), e_1), e_2) \dots, e_n)$. We then have:

Lemma 1. *For any valuation $v \in \mathbb{Q}^P$ and edges $e_1, \dots, e_n$, if we note $(l, Z) = \mathsf{Succ}(\mathsf{Init}(\mathcal{A}), e_1 \dots e_n)$: $v \in Z_{|P}$ iff $\exists \rho \in \mathsf{Runs}(v(\mathcal{A}))$ s.t. $\mathsf{Edges}(\rho) = e_1 \dots e_n$*

4.2 Semi-algorithms for the General Synthesis Problems

The following two algorithms also are natural extensions of their timed automata counterpart. The difficulty here is the handling of the parameter valuations. For $S = (l, Z)$, when non-ambiguous, we use S in place of l or Z to simplify the writing a bit.

For EF we aggregate the valuations found when reaching the locations in G:

$$\mathsf{EF}_G(S, M) = \begin{cases} S_{|P} & \text{if } S \in G \\ \emptyset & \text{if } S \in M \\ \bigcup_{\substack{e \in E \\ S' = \mathsf{Succ}(S, e)}} \mathsf{EF}_G\big(S', M \cup \{S\}\big) & \text{otherwise} \end{cases}$$

For AF, when a path reaches G we retain all valuations that also preserve the other paths that reach G (hence the intersection). If a path cannot reach G we cut it by keeping valuations that make it impossible (in the complement of the projection on the parameters).

$$\mathsf{AF}_G(S,M) = \begin{cases} S_{|P} & \text{if } S \in G \\ \emptyset & \text{if } S \in M \\ \bigcup_{\substack{e\in E \\ S'=\mathsf{Succ}(S,e)}} \mathsf{AF}_G\big(S', M\cup\{S\}\big) \cap \bigcap_{\substack{e'\in E \\ e'\neq e \\ S''=\mathsf{Succ}(S,e')}} \mathsf{AF}_G\big(S'', M\cup\{S\}\big) \cup (\mathbb{R}^P \setminus S''_{|P}) & \text{otherwise} \end{cases}$$

In both algorithms, conditions are evaluated from top to bottom and M represents a *passed list* of symbolic states. It records the symbolic states that have already been explored on a given path. It is possible to have a global passed list shared between all paths but this complicates the writing of the algorithms, especially AF. Initially, M is empty and, for the EF-synthesis problem, for instance, the invocation of EF is, for the PTA $\mathcal{A}$ and a subset of its locations G: $\mathsf{EF}_G(\mathsf{Init}(\mathcal{A}), \emptyset)$.

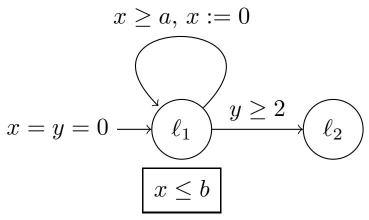


Fig. 1. The PTA $\mathcal{A}_1$ with clocks x and y and parameters a and b

The following theorem states that EF and AF are semi-algorithms for their respective synthesis problems.

Theorem 3. *For any PTA $\mathcal{A}$ and any subset of its locations G, upon termination, $\mathsf{EF}_G(\mathsf{Init}(\mathcal{A}), \emptyset)$ (resp. $\mathsf{AF}_G(\mathsf{Init}(\mathcal{A}), \emptyset)$) is the solution to the EF-synthesis (resp. AF-synthesis) problem for PTA $\mathcal{A}$ and set of locations to reach G.*

Example 1. In the PTA $\mathcal{A}_1$ in Figure 1, after $n > 0$ iterations of the loop, we get the following valuation set $Z_n = \{0 \leq x \leq b, 0 \leq y, a \leq b, 0 \leq na \leq y - x \leq (n+1)b\}$. We can see that we will never have $Z_m = Z_n$ for $m \neq n$ and therefore neither $\mathsf{EF}_{\{\ell_2\}}(\mathsf{Init}(\mathcal{A}_1), \emptyset)$ nor $\mathsf{AF}_{\{\ell_2\}}(\mathsf{Init}(\mathcal{A}_1), \emptyset)$ will terminate.

4.3 Extension for the Integer Synthesis Problems

We now modify the two semi-algorithms to symbolically compute integer valuations. For that we use the notion of integer hull.

Let $n \in \mathbb{N}$ and let Y be a subset of $\mathbb{R}^n$. We denote by $\mathsf{Conv}(Y)$ the *convex hull* of Y, *i.e.* the smallest convex set containing Y. $\mathsf{IntVects}(Y)$ denotes the subset of all elements of Y with integer coordinates. We call those elements *integer valuations* (or vectors). The *integer hull* of Y, denoted by $\mathsf{IntHull}(Y)$ is the smallest convex set containing all the integer vectors of Y, *i.e.* $\mathsf{IntHull}(Y) = \mathsf{Conv}(\mathsf{IntVects}(Y))$.

We extend $\mathsf{IntVects}$ to symbolic states by: $\mathsf{IntVects}((l, Z)) = (l, \mathsf{IntVects}(Z))$ and extend likewise all the other operators on valuation sets.

We make the following assumption on the symbolic states of PTA.

Assumption 1. *Any non-empty symbolic state computed through the* Succ *operator contains at least one integer point.*

Though there exist pathological PTA for which this is not true, we believe that this is not a severe restriction in practice. For instance, considering only non-strict constraints as invariants, still possibly with strict guards, is an easy restriction that is enough to ensure this property. In any case, since we will compute the polyhedra and their integer hulls, Assumption 1 can be verified on-the-fly, at a very low additional cost, during the computation. Under this assumption, we now show that to address our integer parametric problems, it is sufficient to consider the integer hulls of the (valuations in the) symbolic states.

We therefore consider the semi-algorithm IEF (resp. IAF) obtained from EF (resp. AF) by replacing all occurrences of the operator Succ by ISucc with $\mathsf{ISucc}((l,Z),e) = \mathsf{IntHull}(\mathsf{Succ}(l,Z),e)$. We also extend ISucc to edge sequences in the same way as for Succ.

To prove the correctness of these two new algorithms, we rely on Lemma 2 that is the equivalent in our integer setting of Lemma 1:

Lemma 2. *For any integer valuation $v \in \mathbb{Z}^P$ and edges $e_1, \ldots, e_n$, if $(l,Z) = \mathsf{ISucc}(\mathsf{Init}(\mathcal{A}), e_1 \ldots e_n)$: $v \in Z_{|P}$ iff $\exists \rho \in \mathsf{Runs}(v(\mathcal{A}))$ s.t. $\mathsf{Edges}(\rho) = e_1 \ldots e_n$*

Finally, we can state the main result of this subsection: IEF and IAF are correct semi-algorithms for their respective *integer* synthesis problems.

Theorem 4. *For any PTA $\mathcal{A}$ and any subset of its locations G, upon termination, $\mathsf{IEF}_G(\mathsf{Init}(\mathcal{A}),\emptyset)$ (resp. $\mathsf{IAF}_G(\mathsf{Init}(\mathcal{A}),\emptyset)$) is the solution to the integer EF-synthesis (resp. AF-synthesis) problem for PTA $\mathcal{A}$ and set of locations to reach G.*

Example 2. Let us go back to the PTA $\mathcal{A}_1$ in Figure 1. After n iterations of the loop, we still get the same valuation set $Z_n = \{0 \leq x \leq b, 0 \leq y, \leq na \leq y - x \leq (n+1)b\}$. This is because Z_n is its own integer hull. So, again neither $\mathsf{IEF}_{\{\ell_2\}}(\mathsf{Init}(\mathcal{A}_1),\emptyset)$ nor $\mathsf{IAF}_{\{\ell_2\}}(\mathsf{Init}(\mathcal{A}_1),\emptyset)$ will terminate.

4.4 Termination for the Bounded Integer Synthesis Problems

To ensure termination of semi-algorithms IEF and IAF, we now consider that we are searching for bounded integer parameter valuations, *i.e.*, given *a priori* some $M, N \in \mathbb{N}$, we search for valuations in $[-M..N]^P$. Again, this induces new emptiness and synthesis problems that we call (M,N)-*bounded integer* problems (e.g., $(100,100)$-bounded integer EF-emptiness problem).

First remark that, in a TA with $|L|$ locations and $R(m)$ regions (m being the maximal constant appearing in the constraints of the TA), if some location ℓ is reachable, then there exists a run that leads to ℓ and visits at most $|L| \times R(m)$ states. Since it takes at most 1 time unit to go from one region to another, the duration of this run is at most $|L| \times R(m)$ time units. So, if we add invariants $x \leq |L| \times R(m)$ for all clocks x in all the locations of the TA, we obtain an equivalent TA, with respect to location reachability and unavoidability. Since $R(m)$ is non-decreasing with m, this is also true if we increase the value of m.

Now, in our bounded integer parameters setting, we can compute a constant upper bound for each parametric linear expression used in the guards and invariants of the automaton. Let K be the maximum of those upper bounds and of the constants in the non-parametric constraints of the TA. Using the reasoning above, we can then add for all clocks x the invariant $x \leq |L| \times R(K)$ to all locations of our PTA and obtain an equivalent PTA, with respect to location reachability and unavoidability.

For such a PTA $\mathcal{A}$ with bounded clocks and for any valuation $v \in [-M..N]^P$, $v(\mathcal{A})$ is a TA with bounded clocks for which the finiteness of the number of zones computed with the Succ operator is thus ensured.

Let us define an extension of the Init operator that accepts a bound on the values of the parameters in the initial symbolic state (and therefore in the whole computation): for any $M, N \in \mathbb{N}$, $\mathsf{Init}_{M,N}(\mathcal{A}) = (l_0, \{v \in \mathbb{R}^{X \cup P} \mid v_{|X} \in \{\mathbf{0}_X\}^{\nearrow} \cap v_{|P}(\mathsf{Inv}(l_0))_{|X}$ and $v_{|P} \in [-M..N]^P\})$.

Theorem 4 can be naturally adapted to this setting in the following form:

Theorem 5. *For any $M, N \in \mathbb{N}$, any PTA $\mathcal{A}$ and any subset of its locations G, upon termination, $\mathsf{IEF}_G(\mathsf{Init}_{M,N}(\mathcal{A}), \emptyset)$ (resp. $\mathsf{IAF}_G(\mathsf{Init}_{M,N}(\mathcal{A}), \emptyset)$) is the solution to the (M, N)-bounded integer EF-synthesis (resp. AF-synthesis) problem for PTA $\mathcal{A}$ and set of locations to reach G.*

To prove the termination of our computations, we rely on Lemma 3, which states that computing the integer hull of a symbolic state is equivalent to separately computing each of its subsets corresponding to integer parameters and then taking the convex hull of their union.

Lemma 3. *For any symbolic state (l, Z) of the PTA $\mathcal{A}$ s.t. $\forall v \in \mathsf{IntVects}(Z_{|P})$, $v(Z)$ is convex and has integer vertices:* $\mathsf{IntHull}(Z) = \mathsf{Conv}(\bigcup_{v \in \mathsf{IntVects}(Z_{|P})} v(Z))$

We can finally prove that, in this setting, the semi-algorithms do terminate:

Theorem 6. *For any $M, N \in \mathbb{N}$, any PTA $\mathcal{A}$ and any subset of its locations G, Algorithms $\mathsf{IEF}_G(\mathsf{Init}_{M,N}(\mathcal{A}), \emptyset)$ and $\mathsf{IAF}_G(\mathsf{Init}_{M,N}(\mathcal{A}), \emptyset)$ terminate.*

Example 3. Consider once again the PTA $\mathcal{A}_1$ in Figure 1. We now suppose that both parameters are bounded and take their values, say in $[0..3]$. Then as seen above, we add the invariants $x \leq 4$ and $y \leq 4$ to both locations (4 is less than the bound proposed above but enough in this simple case and keeps the computation understandable). This preserves location-based reachability and unavoidability properties. Now, after $n > 0$ iterations of the loop with the "normal" Succ operator, we have the valuation set $Z_n = \{0 \leq a \leq 3, 0 \leq b \leq 3, a \leq b, 0 \leq x \leq 4, 0 \leq y \leq 4, x \leq b, na \leq y - x \leq (n+1)b\}$. If we do not suppose that a and b are integers, we still never have $Z_m = Z_n$ for any $m \neq n$. If we do suppose they are integers, we compute each time $Z'_n = \mathsf{IntHull}(Z_n)$. We have $Z'_0 = Z_0$, $Z'_1 = Z_1 \cap \{y \leq a+3, y \leq b+2\}$, $Z'_2 = Z_2 \cap \{x \leq b-2a+2, y \leq a+3, y-x \leq a+2\}$, $Z'_3 = Z_3 \cap \{a \leq 1, y \leq a+3, y \leq b+3a\}$, $Z'_4 = Z_4 \cap \{y - x = 4a, x \leq 3 - 3a, x \leq b - a\}$, and when $n \geq 5$, $Z'_n = Z'_{n+1} = \{a = 0, x = y, 0 \leq x \leq b, b \leq 3\}$.

And therefore $\mathsf{IEF}_{\{\ell_2\}}(\mathsf{Init}_{0,3}(\mathcal{A}_1), \emptyset)$ terminates and its result is $b \in [1..3]$. Similarly, $\mathsf{IAF}_{\{\ell_2\}}(\mathsf{Init}_{0,3}(\mathcal{A}_1), \emptyset)$ terminates and its result is $a \in [1..3]$ and $b \in [a..3]$.

5 Complexity of the Integer Parametric Problems

When the possible values of the parameters are integer and bounded, we can enumerate all of the possible valuations in exponential time. And therefore, for all classes of problems $\mathcal{P}$ that are EXPTIME for TA, the $\mathcal{P}$-synthesis problem (and of course the $\mathcal{P}$-emptiness) can be solved in exponential time. Also, since the $\mathcal{P}$ problem for TA is always a special case of the $\mathcal{P}$-emptiness problem for PTA, for problems that are complete for some complexity class containing EXPTIME, we can deduce that the corresponding bounded integer emptiness problem is complete for the same complexity class. For instance, the reachability control problem is EXPTIME-complete for TA [13]. The corresponding parametric emptiness problem is: for a PTA $\mathcal{A}$ with actions partitioned between controllable and uncontrollable, does there exist a parameter valuation v such that there exists a controller for $v(\mathcal{A})$ that enforces the reachability of some location whatever the uncontrollable actions that occur? This problem is therefore EXPTIME-complete for bounded integer parameters.

For simpler problems, we have a better and a bit surprising result, using the classical construction of Savitch giving PSPACE=NPSPACE [18]:

Theorem 7. *The $\mathcal{P}$-emptiness problem for PTA with bounded integer parameters is PSPACE-complete for any class of problems $\mathcal{P}$ that is PSPACE-complete for TA.*

In particular the whole TCTL model-checking, including reachability and unavoidability, is PSPACE-complete for TA [1] and as a consequence, the corresponding emptiness problem, which includes EF-emptiness and AF-emptiness, is PSPACE-complete for PTA with bounded integer parameters.

Finally, it is important to remark that we cannot easily lift either of the boundedness or the integer assumptions: the EF-emptiness problem for PTA with *bounded rational* parameter values is undecidable [16], and Theorem 8 follows from the undecidability proof of [3]:

Theorem 8. *The EF-emptiness problem for PTA with* possibly unbounded integer *parameter values is undecidable.*

6 On Performance in Practice: Task Set Schedulability

The PTA in Figure 1 demonstrates that it is very easy to find an example for which the symbolic computation does not terminate without the bounded integer parameters restriction but one could object that this PTA models nothing real (if $a = 0$, there are zeno runs for instance). We now show with a very simple but realistic case-study that this restriction is also useful for real applications.

Consider the scheduling problem adapted from [8] for a non-preemptive setting: we have three real-time tasks τ_1, τ_2 and τ_3. τ_1 is periodic with period a and has an execution time $C_1 \in [10, b]$. τ_2 is sporadic: it has only a minimal delay between two activations and that delay is $2a$. The execution time of τ_2 is $C_2 \in [c, d]$, with $c \leq d$. Finally, τ_3 is periodic with period $3a$ and has an execution time $C_3 \in [20, 28]$. These three tasks are scheduled using a non-preemptive[1] priority policy defined by $\tau_1 > \tau_2 > \tau_3$. We say that the system is schedulable if each task always has at most one instance running, which is a safety property.

We can model this problem with a parametric time Petri net (which, as it will be bounded by the property can be seen as a subclass of PTA [19]) in Roméo. Schedulability is verified using implementations of the semi-algorithms EF and IEF presented in section 4. The symbolic computations use the state class abstraction of [5,19], which is specific to time Petri nets, and do not require extrapolation. It is however very similar to the zone-based abstraction and trivially satisfies Property 1. The rest of the results carry over to that abstraction without any difficulty. We use a machine with an Intel Core I7 at 2.3 GHz and 8 Gb RAM.

Table 1 provides some insight on the performance of Algorithm IEF and a comparison to Algorithm EF. The only difference in the implementations of the two algorithms is the application or not of the integer hull operator. The table shows the total time for the verifications, the part of it used for computing the integer hull for Algorithm IEF, and the memory consumptions. DNF means that the computation did not finish within 90 min (memory was not a problem here). The constraint generated for the first column is $a \geq 44$, for the second $a - b \geq 24, b \geq 10$, and for the third $a - b \geq 24, b \geq 10, 0 \leq c \leq 28$. For the fourth column the constraint is much more complex so we will not reproduce it here. Note that in all these cases some parameters are unbounded so an explicit enumeration of all possible parameter values coupled with an efficient (DBM-based or discrete-time decision diagram-based) verification was not possible (and termination of Algorithm IEF was actually not guaranteed).

Table 1. Usefulness of the Integer Hull

	$a \in [0, \infty)$ $b = 20$ $c = 18$ $d = 28$	$a \in [0, \infty)$ $b \in [10, \infty)$ $c = 18$ $d = 28$	$a \in [0, \infty)$ $b \in [10, \infty)$ $c \in [0, 28]$ $d = 28$	$a \in [0, \infty)$ $b \in [10, \infty)$ $c = 18$ $d \in [18, \infty)$	$a \in [0, \infty)$ $b \in [10, \infty)$ $c \geq 0$ $d \geq c$
IEF Time	1 s	2.8 s	27 s	840 s	DNF
Int. Hull	0.2 s (20%)	0.4 s (14%)	2.9 s (11%)	146 s (17%)	–
IEF Mem.	15 Mo	35 Mo	153 Mo	1289 Mo	–
EF Time	1.5 s	6.4 s	DNF	DNF	DNF
EF Mem.	19.6 Mo	55 Mo	–	–	–

[1] A running task cannot be interrupted even if another task with a greater priority is ready.

Table 2. Scaling a's upper bound for $b \in [10, 100], c = 18$ and $d \in [18, 100]$

	$a \in [0, 100]$	$a \in [0, 1000]$	$a \in [0, 10000]$
IEF Time	1079 s	1150 s	1178 s
Int. Hull	166 s (15.4%)	167 s (14.5%)	168 s (14.3%)
IEF Mem.	1598 Mo	1667 Mo	1667 Mo

With Table 2 we illustrate the smooth scaling of our approach with the value of upper bounds. Not that the performance of Algorithm IEF is actually worse when all parameters are bounded (compare with the fourth column of Table 1). This is due to the fact that our implementation uses inclusion for convergence, which is favored by the reduced number of constraints in the absence of upper bounds. In this setting, termination is guaranteed however.

7 Conclusion

We have presented novel results for the parametric verification of timed systems modeled as parametric timed automata. Our new negative results show that even when severely restricting the form of the parametric constraints we encounter undecidabilty for many interesting problems. So we have proposed instead to restrict the codomain of the valuations to bounded integers.

This is completely orthogonal to previous restriction schemes in the sense that it does not enforce any syntactic restriction on PTA, thus simplifying the modeling activity. Also experimental evidence shows that the symbolic approach we propose to avoid an explicit enumeration of all the possible parameter values is robust to scaling the bounds of the parameters (and improves on convergence even without any bounds in some cases).

Also, in this setting, most problems are of course decidable and we have proved that, for instance, emptiness for TCTL properties, which include reachability and unavoidability, is PSPACE-complete. We have also proved that lifting the boundedness or the integer assumption leads to undecidability. We have exhibited symbolic algorithms that allow to avoid the explicit enumeration of all possible valuations and implemented them in our tool Roméo [15].

Our current lines of work on this problem include improving the computation of the integer hulls, the search for less restrictive codomains for parameter valuations, and extension of this work for parametric timed games and PTA with stopwatches.

Acknowledgments. The authors thank Claude Jard and Enea Zaffanella for the related discussions, as well as an anonymous reviewer from CONCUR'12 for her/his very useful comments.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and Computation 104(1), 2–34 (1993)
2. Alur, R., Dill, D.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: ACM Symposium on Theory of Computing, pp. 592–601 (1993)
4. André, E., Chatain, T., Encrenaz, E., Fribourg, L.: An inverse method for parametric timed automata. In: RP Workshop on Reachability Problems, Liverpool, U.K., vol. 223, pp. 29–46 (2008)
5. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. on Soft. Eng. 17(3), 259–273 (1991)
6. Bozzelli, L., La Torre, S.: Decision problems for lower/upper bound parametric timed automata. Formal Methods in System Design 35(2), 121–151 (2009)
7. Bruyère, V., Raskin, J.-F.: Real-time model-checking: Parameters everywhere. Logical Methods in Computer Science 3(1), 1–30 (2007)
8. Bucci, G., Fedeli, A., Sassoli, L., Vicario, E.: Time state space analysis of real-time preemptive systems. IEEE Trans. on Soft. Eng. 30(2), 97–111 (2004)
9. Doyen, L.: Robust parametric reachability for timed automata. Information Processing Letters 102(5), 208–213 (2007)
10. Henzinger, T.A., Ho, P.-H., Wong-toi, H.: Hytech: A model checker for hybrid systems. Software Tools for Technology Transfer 1, 460–463 (1997)
11. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inform. and Computation 111(2), 193–244 (1994)
12. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.: Linear parametric model checking of timed automata. Journal of Logic and Algebraic Programming 52-53, 183–220 (2002)
13. Jurdziński, M., Trivedi, A.: Reachability-Time Games on Timed Automata. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 838–849. Springer, Heidelberg (2007)
14. Larsen, K.G., Pettersson, P., Yi, W.: Model-Checking for Real-Time Systems. In: Reichel, H. (ed.) FCT 1995. LNCS, vol. 965, pp. 62–88. Springer, Heidelberg (1995)
15. Lime, D., Roux, O.H., Seidner, C., Traonouez, L.-M.: Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 54–57. Springer, Heidelberg (2009)
16. Miller, J.S.: Decidability and Complexity Results for Timed Automata and Semi-linear Hybrid Automata. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 296–310. Springer, Heidelberg (2000)
17. Minsky, M.: Computation: Finite and Infinite Machines. Prentice Hall (1967)
18. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. 4(2), 177–192 (1970)
19. Traonouez, L.-M., Lime, D., Roux, O.H.: Parametric model-checking of stopwatch Petri nets. Journal of Universal Computer Science 15(17), 3273–3304 (2009)
20. Virbitskaite, I., Pokozy, E.: Parametric Behaviour Analysis for Time Petri Nets. In: Malyshkin, V.E. (ed.) PaCT 1999. LNCS, vol. 1662, pp. 134–140. Springer, Heidelberg (1999)
21. Wang, F.: Parametric timing analysis for real-time systems. Information and Computation 130(2), 131–150 (1996)

LTL Model-Checking for Malware Detection*

Fu Song and Tayssir Touili

LIAFA, CNRS and Univ. Paris Diderot, France
{song,touili}@liafa.univ-paris-diderot.fr

Abstract. Nowadays, malware has become a critical security threat. Traditional anti-viruses such as signature-based techniques and code emulation become insufficient and easy to get around. Thus, it is important to have efficient and robust malware detectors. In [20,19], CTL model-checking for PushDown Systems (PDSs) was shown to be a robust technique for malware detection. However, the approach of [20,19] lacks precision and runs out of memory in several cases. In this work, we show that several malware specifications could be expressed in a more precise manner using LTL instead of CTL. Moreover, LTL can express malicious behaviors that cannot be expressed in CTL. Thus, since LTL model-checking for PDSs is polynomial in the size of PDSs while CTL model-checking for PDSs is exponential, we propose to use LTL model-checking for PDSs for malware detection. Our approach consists of: (1) Modeling the binary program as a PDS. This allows to track the program's stack (needed for malware detection). (2) Introducing a new logic (SLTPL) to specify the malicious behaviors. SLTPL is an extension of LTL with variables, quantifiers, and predicates over the stack. (3) Reducing the malware detection problem to SLTPL model-checking for PDSs. We reduce this model checking problem to the emptiness problem of Symbolic Büchi PDSs. We implemented our techniques in a tool, and we applied it to detect several viruses. Our results are encouraging.

1 Introduction

Over the past decades, the landscape of malware's intent has changed. More and more sophisticated malwares have been designed for more general cyber-espionage purposes. For example, Stuxnet, Duqu and Flame are deployed for targeted attacks in countries, such as Iran, Israel, Sudan. Traditional antivirus techniques: code emulation and signature (pattern)-based techniques become insufficient. Indeed, code emulation techniques monitoring only several traces of programs in a limited time span may miss some malicious behaviors, and signature-based techniques using patterns of programs' codes to characterize malware can only detect known malwares.

Addressing these limitations, many efforts have been made [1,4,5,17,7,8,13,2]. Among them, model-checking is one of the efficient techniques for malware detection [4,17,7,8,13], as it allows to check the behavior (not the syntax) of the program without executing it. However, [4,17,7,8,13] use finite state graphs (automata) as program model that cannot accurately represent the program's stack. Being able to track the program's stack is very important for malware detection as explained in [16]. For example,

* Work partially funded by ANR grant ANR-08-SEGI-006.

malware writers obfuscate the system calls by using pushes and jumps to make malware hard to analyze, because anti-viruses usually determine malware by checking function calls to operating systems.

To overcome this problem, we proposed a new approach for malware detection in [20,19] that consists of (1) Modeling the program using a Pushdown System (PDS). This allows us to track the behavior of the stack. (2) Introducing a new logic, called SCTPL, to specify malicious behaviors. SCTPL can be seen as an extension of the branching-time temporal logic CTL with variables, quantifiers, and regular predicates over the stack. Extension with variables and quantifiers allows to express malicious behaviors in a more succinct way and regular predicates allow to specify properties on the stack content which is important for malware detection. (3) And reducing the malware detection problem to the model-checking problem of PDSs against SCTPL formulas. Our techniques were implemented and applied to detect several viruses.

However, using the techniques of [20,19], the analysis of several malwares runs out of memory due to the complexity of SCTPL model-checking for PDSs. By looking carefully at the SCTPL formulas specifying the malicious behaviors, we found that most of these SCTPL formulas can be expressed in a more precise manner using the Linear Temporal Logic (LTL). Since LTL can express some malicious behaviors that cannot be expressed by SCTPL, and since the complexity of LTL model-checking for PDSs is polynomial in the size of PDSs, whereas the complexity of CTL model-checking for PDSs is exponential, we will apply in this work LTL model-checking for malware detection (instead of applying SCTPL model-checking as we did in [20,19], since this technique lacks precision and runs out of memory in several cases.). To obtain succinct LTL formulas that express malicious behaviors, we follow the idea of [20,19] and introduce the SLTPL logic, an extension of LTL with variables, quantifiers and regular predicates over the stack content. SLTPL is as expressive as LTL with regular valuations [9,14], but it allows to express malicious behaviors in a more succinct way. We show that SLTPL model-checking for PDSs is polynomial in the size of PDSs and we reduce the malware detection problem to SLTPL model-checking for PDSs.

We use the approach of [19] to model a program as a PDS, in which the PDS control locations correspond to the program's control points, and the PDS's stack mimics the program's execution stack. This approach allows to track the program's stack.

In SLTPL, propositions can be predicates of the form $p(x_1, \ldots, x_n)$, where the x_i's are free variables or constants. Free variables can get their values from a finite domain. Variables can be universally or existentially quantified. SLTPL without predicates over the stack content (called LTPL) is as expressive as LTL, but it allows to express malicious behaviors in a more succinct way. For example, consider the statement "There is a register assigned by 0, and then, the content of this register is pushed onto the stack." This statement can be expressed in LTL as a large formula enumerating all the possible registers as follows:

$$\Big(mov(eax,0) \wedge \mathbf{X}\, push(eax)\Big) \vee \Big(mov(ebx,0) \wedge \mathbf{X}\, push(ebx)\Big) \vee \Big(mov(ecx,0) \wedge \mathbf{X}\, push(ecx)\Big) \vee \ldots$$

where every instruction is regarded as a predicate, e.g., $mov(eax,0)$ is a predicate. However, this LTL formula is large for such a simple statement. Using LTPL, this can be expressed by $\exists r\ (mov(r,0) \wedge \mathbf{X}\, push(r))$ which expresses in a succinct way that there exists a *register* r s.t. the above holds.

However, LTPL cannot specify properties about the stack, which is important for malware detection as explained previously. For example, consider Fig. 1(a). It corresponds to a critical fragment of the Trojan LdPinch that adds itself into the registry key listing to get started at boot time. To do this, it calls the API function *GetModuleFileNameA* with 0 and an address a as parameters [1]. After calling this function, the file name of its own executable will be stored in the address a. Then, the API function *RegSetValueExA* is called with a as parameter (i.e., its own file name). This adds its file name into the registry key listing. We cannot specify this malicious behavior in a precise manner using LTPL. Indeed, a virus writer can easily use some obfuscation techniques in order to escape from any LTPL specification. E.g., let us introduce one *push* followed by one *pop* after *push 0* at line l_2 as done in Fig. 1(b). This fragment has the same malicious behavior than the fragment in Fig. 1(a). Since the number of pushes and pops can be arbitrary, it is always possible for virus writers to change their code in order to escape from a given LTPL formula. To overcome this problem, we introduce SLTPL, which is extension of LTPL with regular predicates over the stack. Such predicates are given by Regular Variable Expressions over the stack alphabet and some free variables (which can also be existentially and universally quantified). SLTPL is as expressive as LTL with regular valuations [9], but more succinct. In this setting, the malicious behavior of Fig. 1(a) and (b) can be specified as follows: $\mathbf{F}\ \exists a\Big(call(GetModuleFileNameA) \wedge 0\ a\ \Gamma^* \wedge \mathbf{F}(call(RegSetValueExA) \wedge a\ \Gamma^*)\Big)$, where $0\ a\ \Gamma^*$ (resp. $a\ \Gamma^*$) is a predicate expressing that the top of the stack are 0 and a (resp. a). The SLTPL formula states that there exists a path in which *GetModuleFileNameA* is called with 0 and some address a as parameters (i.e., 0 and a are on the top of the stack), later *RegSetValueExA* is called with a as parameter. This specification can detect both fragments in Fig. 1(a) and (b), because it allows to specify the content of the stack when *GetModuleFileNameA* is called. Note that it is important to use PDSs as a model in order to have specifications with predicates over the stack.

```
l1: push a
l2: push 0
l3: call GetModuleFileNameA
l4: push a
l5: call RegSetValueExA
```
(a)

```
l'1: push a
l'2: push 0
l'3: push eax
l'4: pop eax
l'5: call GetModuleFileNameA
l'6: push a
l'7: call RegSetValueExA
```
(b)

Fig. 1. (a) A fragment of the Trojan LdPinch and (b) The obfuscated version

Thus, we reduce the malware detection problem to the SLTPL model checking problem for PDSs. To solve this problem, we first present a reduction from LTPL model-checking for PDSs to the emptiness problem of Symbolic Büchi PDSs (SBPDS). This latter problem can be efficiently solved by [10]. Then, we consider the SLTPL model checking problem for PDSs. We introduce Extended Finite Automata (EFA) to represent regular predicates. To perform SLTPL model-checking, we first construct a *Symbolic PDS* which is a kind of synchronization of the PDS and the EFAs that allows to determine whether the stack predicates hold at a given step by looking only at the top

[1] Parameters to a function in assembly are passed by pushing them onto the stack before a call to the function is made. The code in the called function later retrieves these parameters from the stack.

of the stack of the symbolic PDS. This allows us to reduce the SLTPL model-checking problem for PDSs to the emptiness problem of SBPDSs.

We implemented our techniques in a tool and applied it to detect malwares. Our tool can detect all the malwares that we considered. The experimental results show that detecting malware using SLTPL model-checking performs better than using SCTPL model-checking [20,19] and LTL model-checking for PDSs with regular valuations [9]. Moreover, the analysis of several examples terminated using SLTPL model-checking, while it runs out of memory/time using SCTPL or LTL with regular valuations model-checking. Moreover, some malicious behaviors as expressed in [20,19] produce some false alarms. Using SLTPL, these false alarms are avoided. Our tool can also detect the notorious malware *Flame* that was undetected for more than five years.

Related Work: Quantified Linear Temporal Logic (QLTL) [18] is close to LTPL. However, QLTL disallows to quantify over atomic propositions' parameters. LTPL is a subclass of the First-order Linear Temporal Logic (FO-LTL) [12]. [12] does not consider the model-checking problem. $\mathcal{L}_{\mathcal{MDG}}$ [22] and $\mathcal{L}_{\mathcal{MDG}^*}$ [21] are sub-logics of FO-LTL. However, $\mathcal{L}_{\mathcal{MDG}}$ disallows temporal operator nesting and properties beyond its templates, and $\mathcal{L}_{\mathcal{MDG}^*}$ cannot use existential and universal operators. FO-LTL was used for malware detection in [3]. All these works cannot specify predicates over the stack.

Model-checking and static analysis such as [4,17,7,8,13,1,2] have been applied to detect malicious behaviors. However, all these works are based on modeling the program as a finite-state system, and thus, they miss the behavior of the stack. As explained in the introduction, being able to track the stack is important for many malicious behaviors. [16] keeps track of the stack by computing an abstract stack graph which finitely represents the infinite set of all the possible stacks for every control point of the program. Their technique can detect some malicious behaviors that change the stack. However, they cannot specify the other malicious behaviors that SLTPL can describe. [15] performs context-sensitive analysis of *call* and *ret* obfuscated binaries. They use abstract interpretation to compute an abstraction of the stack. We believe that our techniques are more precise since we do not abstract the stack. Moreover, the techniques of [15] were only tried on toy examples, they have not been applied for malware detection.

CTPL [13] is an extension of CTL with variables and quantifiers. SCTPL [20,19] is an extension of CTPL with predicates over the stack content. CTL, CTPL and SCTPL are incomparable with LTPL or SLTPL. For malware detection, experimental results show that SLTPL model-checking performs better and is more precise.

Outline. Sections 2 and 3 give the definition of PDSs and LTPL/SLTPL, respectively. LTPL/SLTPL model-checking for PDSs are given in Sections 4 and 5, respectively. Experiments are shown in Section 6.

2 Binary Code Modeling

In this section, we recall the definition of pushdown systems. We use the translation of [19] to model binary programs as pushdown systems.

A *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the stack alphabet, and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules. A configuration of $\mathcal{P}$ is $\langle p, \omega \rangle$, where $p \in P$ and $\omega \in \Gamma^*$. If $((p,\gamma),(q,\omega)) \in \Delta$, we write $\langle p,\gamma \rangle \hookrightarrow \langle q,\omega \rangle$ instead. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: for every $\omega' \in \Gamma^*$, $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ if $\langle p,\gamma \rangle \hookrightarrow \langle q,\omega \rangle$. For every configuration $c, c' \in P \times \Gamma^*$, c' is an immediate successor of c iff $c \rightsquigarrow_{\mathcal{P}} c'$. An execution of $\mathcal{P}$ is a sequence of configurations $\pi = c_0 c_1 ...$ s.t. $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$ for every $i \geq 0$. Let $\pi(i)$ denote c_i and π^i denote the *suffix* of π starting from $\pi(i)$. For technical reasons, w.l.o.g., we assume that for every transition rule $\langle p,\gamma \rangle \hookrightarrow \langle q,\omega \rangle$, $|\omega| \leq 2$ (see [10]).

3 Malicious Behavior Specification

We define the Stack Linear Temporal Predicate Logic (SLTPL) as an extension of the Linear Temporal Logic (LTL) with variables and regular predicates over the stack content. Variables are parameters of atomic predicates and can be quantified by the existential and universal operators. Regular predicates are represented by regular variable expressions and are used to specify the stack content of the PDS.

3.1 Environments, Predicates and Regular Variable Expressions

From now on, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, ...\}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment that assigns a value $c \in \mathcal{D}$ to each variable $x \in \mathcal{X}$ s.t. $B(c) = c$ for every $c \in \mathcal{D}$. $B[x \leftarrow c]$ denotes the environment s.t. $B[x \leftarrow c](x) = c$ and $B[x \leftarrow c](y) = B(y)$ for every $y \neq x$. Let $\mathcal{B}$ be the set of all the environments. Let $\Theta_{id} = \{(B_1, B_2) \in \mathcal{B} \times \mathcal{B} \mid B_1 = B_2\}$ be the identity relation for environments, and for every $x \in \mathcal{X}$, $\Theta_x = \{(B_1, B_2) \in \mathcal{B} \times \mathcal{B} \mid \forall x' \in \mathcal{X} \textit{ s.t. } x \neq x', B_1(x') = B_2(x')\}$ be the relation that abstracts away the value of x.

Let $AP = \{a, b, c, ...\}$ be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates of the form $b(\alpha_1, ..., \alpha_m)$ s.t. $b \in AP$, $\alpha_i \in \mathcal{X} \cup \mathcal{D}$ for every i, $1 \leq i \leq m$, and $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $b(\alpha_1, ..., \alpha_m)$ s.t. $b \in AP$ and $\alpha_i \in \mathcal{D}$ for every i, $1 \leq i \leq m$.

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, a finite set $\mathcal{R}$ of *Regular Variable Expressions* (RVEs) e over $\mathcal{X} \cup \Gamma$ is defined by: $e ::= \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e \cdot e \mid e^*$. The language $L(e)$ of a RVE e is a subset of $P \times \Gamma^* \times \mathcal{B}$ defined inductively as follows: $L(\epsilon) = \{(\langle p, \epsilon \rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(x)$, where $x \in \mathcal{X}$ is the set $\{(\langle p, \gamma \rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(x) = \gamma\}$; $L(\gamma)$, where $\gamma \in \Gamma$ is the set $\{(\langle p, \gamma \rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(e_1 + e_2) = L(e_1) \cup L(e_2)$; $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2 \rangle, B) \mid (\langle p, \omega_1 \rangle, B) \in L(e_1);\ (\langle p, \omega_2 \rangle, B) \in L(e_2)\}$; $L(e^*) = \{(\langle p, \omega \rangle, B) \mid \omega \in \{u \in \Gamma^* \mid (\langle p, u \rangle, B) \in L(e)\}^*\}$.

3.2 The Stack Linear Temporal Predicate Logic

A SLTPL formula is a LTL formula where predicates and RVEs are used as atomic propositions, and where quantifiers over variables are used. For technical reasons, we suppose w.l.o.g. that formulas are given in positive normal form. We use the *release operator* **R** as the dual of the until operator **U**. Formally, the set of SLTPL formulas is given by (where $x \in \mathcal{X}$, $e \in \mathcal{R}$ and $b(\alpha_1, ..., \alpha_m) \in AP_{\mathcal{X}}$):

$$\varphi ::= b(\alpha_1, ..., \alpha_m) \mid \neg b(\alpha_1, ..., \alpha_m) \mid e \mid \neg e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\, \varphi \mid \exists x\, \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi \mid \varphi \mathbf{R} \varphi$$

The other standard operators of LTL can be expressed by the above operators: $\mathbf{F}\psi = true\mathbf{U}\psi$ and $\mathbf{G}\psi = false\mathbf{R}\psi$. A SLTPL formula ψ is a LTPL formula iff the formula ψ does not use any regular predicate $e \in \mathcal{R}$. A variable x is a *free variable* of ψ if it is out of the scope of a quantification in ψ.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\lambda : AP_{\mathcal{D}} \rightarrow 2^P$ be a labeling function that assigns a set of control locations to each predicate. Let $c = \langle p, \omega \rangle$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SLTPL formula ψ in c (denoted by $c \models_\lambda \psi$) iff there exists an environment $B \in \mathcal{B}$ s.t. c satisfies ψ under B (denoted by $c \models_\lambda^B \psi$). $c \models_\lambda^B \psi$ holds iff there exists an execution π starting from c s.t. π satisfies ψ under B (denoted by $\pi \models_\lambda^B \psi$), where $\pi \models_\lambda^B \psi$ is defined by induction as follows: $\pi \models_\lambda^B b(\alpha_1, ..., \alpha_m)$ iff the control location p of $\pi(0)$ is in $\lambda(b(B(\alpha_1), ..., B(\alpha_m)))$; $\pi \models_\lambda^B \neg b(\alpha_1, ..., \alpha_m)$ iff $\pi \models_\lambda^B b(\alpha_1, ..., \alpha_m)$ does not true; $\pi \models_\lambda^B e$ iff $(\pi(0), B) \in L(e)$; $\pi \models_\lambda^B \neg e$ iff $(\pi(0), B) \notin L(e)$; $\pi \models_\lambda^B \psi_1 \wedge \psi_2$ iff $\pi \models_\lambda^B \psi_1$ and $\pi \models_\lambda^B \psi_2$; $\pi \models_\lambda^B \psi_1 \vee \psi_2$ iff $\pi \models_\lambda^B \psi_1$ or $\pi \models_\lambda^B \psi_2$; $\pi \models_\lambda^B \forall x\, \psi$ iff for every $v \in \mathcal{D}$, $\pi \models_\lambda^{B[x \leftarrow v]} \psi$; $\pi \models_\lambda^B \exists x\, \psi$ iff there exists $v \in \mathcal{D}$ s.t. $\pi \models_\lambda^{B[x \leftarrow v]} \psi$; $\pi \models_\lambda^B \mathbf{X}\, \psi$ iff $\pi^1 \models_\lambda^B \psi$; $\pi \models_\lambda^B \psi_1 \mathbf{U} \psi_2$ iff there exists $i \geq 0$ s.t. $\pi^i \models_\lambda^B \psi_2$ and $\forall j,\ 0 \leq j < i : \pi^j \models_\lambda^B \psi_1$; $\pi \models_\lambda^B \psi_1 \mathbf{R} \psi_2$ iff for all $j \geq 0$, if for any $i < j : \pi^i \not\models_\lambda^B \psi_1$, then $\pi^j \models_\lambda^B \psi_2$.

Given a SLTPL formula ψ, let $cl_\exists(\psi)$ (resp. $cl_\forall(\psi)$ and $cl_\mathbf{U}(\psi)$) denote the set of $\exists$-formulas (resp. $\forall$-formulas and $\mathbf{U}$-formulas) of the form $\exists x\phi$ (resp. $\forall x\phi$ and $\phi_1\mathbf{U}\phi_2$) of ψ. Let $cl(\psi)$ be the *closure* of ψ defined as the smallest set of formulas containing ψ and satisfying the following: if $\phi_1 \wedge \phi_2 \in cl(\psi)$ or $\phi_1 \vee \phi_2 \in cl(\psi)$, then $\phi_1 \in cl(\psi)$ and $\phi_2 \in cl(\psi)$; if $\mathbf{X}\phi_1 \in cl(\psi)$, or $\exists x\phi_1 \in cl(\psi)$, or $\forall x\phi_1 \in cl(\psi)$ or $\neg\phi_1 \in cl(\psi)$, then $\phi_1 \in cl(\psi)$; if $\phi_1\mathbf{U}\phi_2 \in cl(\psi)$, then $\phi_1 \in cl(\psi), \phi_2 \in cl(\psi)$ and $\mathbf{X}(\phi_1\mathbf{U}\phi_2) \in cl(\psi)$; if $\phi_1\mathbf{R}\phi_2 \in cl(\psi)$, then $\phi_1 \in cl(\psi), \phi_2 \in cl(\psi)$ and $\mathbf{X}(\phi_1\mathbf{R}\phi_2) \in cl(\psi)$.

LTL with regular valuations is an extension of LTL where the atomic propositions can be regular sets of configurations over the stack alphabet [9,14]. SLTPL is as expressive as LTL with regular valuations. Since the domain $\mathcal{D}$ is finite, we have:

Proposition 1. *LTPL and LTL (resp. SLTPL and LTL with regular valuations) have the same expressive power. SLTPL is more expressive than LTL.*

3.3 Modeling Malicious Behaviors Using SLTPL

We consider a typical malicious behavior: windows viruses that compute the entry address of Kernel32.dll. We show that this behavior can be expressed in a more precise manner using SLTPL instead of SCTPL, and that if we use SCTPL to describe it, we can obtain false alarms that can be avoided when using SLTPL (see Tab. 1).

Kernel32.dll Base Address Viruses:
Many Windows viruses use API functions to achieve their malicious tasks. The Kernel32.dll file includes several API functions that can be used by the viruses. In order to use these functions, the viruses have to find the entry addresses of these API functions. To do this, they need to determine the Kernel32.dll entry point. They determine first the Kernel32.dll PE

```
l1 : cmp [eax], 5A4Dh
jnz l2
  ...
cmp [ebx], 4550h
jz l3
l2 : ...
jmp l1
l3
```

(a)

```
l1' : ...
  ...
jnz l1'
cmp [eax], 5A4Dh
cmp [ebx], 4550h
```

(b)

Fig. 2.

header in memory and use this information to locate the Kernel32.dll export section and find the entry addresses of the API functions. For this, the virus looks first for the DOS header (the first word of the DOS header is *5A4Dh* in hex (*MZ* in ascii)); and then looks for the PE header (the first two words of the PE header is *4550h* in hex (*PE*00 in ascii)). Fig. 2(a) presents a disassembled code fragment performing this malicious behavior. This behavior can be specified in SLTPL using the formula $\Psi_{wv} = \mathbf{GF}(\exists r_1\ cmp(r_1, 5A4Dh) \wedge \mathbf{F}\ \exists r_2\ cmp(r_2, 4550h))$. This SLTPL formula expresses that the program has a loop such that there are two variables r_1 and r_2 such that first, r_1 is compared to *5A4Dh*, and then r_2 is compared to *4550h*. This formula can detect the malware in Fig. 2(a). It can be shown that there is no CTL-like formula equivalent to Ψ_{wv}. In [20,19], to be able to express this malicious behavior using a CTL-like formula, we used the following formula: $\Psi'_{wv} = \mathbf{EGEF}(\exists r_1\ cmp(r_1, 5A4Dh) \wedge \mathbf{EF}\ \exists r_2\ cmp(r_2, 4550h))$. This formula can detect the malware in Fig. 2(a). However, the benign program in Fig. 2(b) that compares with *5A4Dh* and *4550h* only *once* is also detected as a malware using Ψ'_{wv} due to the loop at l'_1, while Ψ_{wv} will not detect it as a malware. In our experiments, as shown in Tab. 1, several benign programs are detected as malwares using Ψ'_{wv}, whereas Ψ_{wv} classified them as benign programs.

4 LTPL Model-Checking for PDSs

In this section, we show how to reduce LTPL model-checking for PDSs to the emptiness problem of symbolic Büchi PDSs which can be efficiently solved by [10].

4.1 Symbolic Büchi Pushdown Systems

A *Symbolic Pushdown System* (SPDS) $\mathcal{P}$ is a tuple (P, Γ, Δ), where P is a finite set of control locations, Γ is the stack alphabet and Δ is a set of symbolic transition rules of the form $\langle p, \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle q, \omega\rangle$ s.t. $p, q \in P, \gamma \in \Gamma, \omega \in \Gamma^*$, and $\Theta \subseteq \mathcal{B} \times \mathcal{B}$.

A *symbolic* transition $\langle p, \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle q, \omega\rangle$ denotes the following set of PDS transition rules: $\langle (p, B), \gamma\rangle \hookrightarrow \langle (q, B'), \omega\rangle$ for every $B, B' \in \mathcal{B}$ s.t. $(B, B') \in \Theta$. For every $\omega' \in \Gamma^*$, $\langle (q, B'), \omega\omega'\rangle$ is an immediate successor of $\langle (p, B), \gamma\omega'\rangle$, denoted by $\langle (p, B), \gamma\omega'\rangle \rightsquigarrow_{\mathcal{P}} \langle (q, B'), \omega\omega'\rangle$. A run (execution) of $\mathcal{P}$ from $\langle (p_0, B_0), \omega_0\rangle$ is a sequence $\langle (p_0, B_0), \omega_0\rangle\langle (p_1, B_1), \omega_1\rangle \cdots$ over $P \times \mathcal{B} \times \Gamma^*$ s.t. for every $i \geq 0$, $\langle (p_i, B_i), \omega_i\rangle \rightsquigarrow_{\mathcal{P}} \langle (p_{i+1}, B_{i+1}), \omega_{i+1}\rangle$.

A *Symbolic Büchi Pushdown System* (SBPDS) $\mathcal{BP}$ is a tuple (P, Γ, Δ, F), where (P, Γ, Δ) is a SPDS and $F \subseteq P$ is a finite set of accepting control locations. A run of the SBPDS $\mathcal{BP}$ is accepting iff it infinitely often visits some control locations in F. Let $L(\mathcal{BP})$ be the set of configurations $\langle (p, B), \omega\rangle \in P \times \mathcal{B} \times \Gamma^*$ from which $\mathcal{BP}$ has an accepting run.

Theorem 1. *Given a SBPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$, for every configuration $\langle (p, B), \omega\rangle \in P \times \mathcal{B} \times \Gamma^*$, whether or not $\langle (p, B), \omega\rangle \in L(\mathcal{BP})$ can be decided in time* $O(|P| \cdot |\Delta|^2 \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.

Given a SBPDS $\mathcal{BP}$ with n control locations, m boolean variables and d transition rules, [10] shows that $L(\mathcal{BP})$ can be computed in time $O(n \cdot 2^{3m} \cdot d^2)$. We can use

$|\mathcal{X}| \cdot \log_2 |\mathcal{D}|$ boolean variables to represent the set of variables $\mathcal{X}$ over $\mathcal{D}$. Thus, we can decide whether $\langle(p, B), \omega\rangle$ is in $L(\mathcal{BP})$ in time $O(|P| \cdot |\Delta|^2 \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.

A *Generalized Symbolic Büchi PDS* (GSBPDS) $\mathcal{BP}$ is a tuple (P, Γ, Δ, F), where (P, Γ, Δ) is a SPDS and $F = \{F_1, ..., F_k\}$ is a set of sets of accepting control locations. A run of the GSBPDS $\mathcal{BP}$ is accepting iff for every i, $1 \le i \le k$, the run infinitely often visits some control locations in F_i. Let $L(\mathcal{BP})$ denote the set of configurations $\langle(p, B), \omega\rangle \in P \times \mathcal{B} \times \Gamma^*$ from which the GSBPDS $\mathcal{BP}$ has an accepting run.

Proposition 2. *Given a GSBPDS $\mathcal{BP}$, we can get a SBPDS $\mathcal{BP}'$ s.t. $L(\mathcal{BP}) = L(\mathcal{BP}')$.*

4.2 From LTPL Model-Checking for PDSs to the Emptiness Problem of SBPDSs

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, $\lambda : AP_{\mathcal{D}} \to 2^P$ a labeling function, ψ a LTPL formula. We construct a GSBPDS $\mathcal{BP}_\psi$ s.t. $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \{\psi\}|\!), B), \omega\rangle$ iff $\mathcal{P}$ has an execution π from $\langle p, \omega\rangle$ s.t. π satisfies ψ under B. Thus, $\langle p, \omega\rangle \models_\lambda \psi$ iff there exists $B \in \mathcal{B}$ s.t. $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \{\psi\}|\!), B), \omega\rangle$ (since $\langle p, \omega\rangle \models_\lambda \psi$ iff there exists $B \in \mathcal{B}$ s.t. $\langle p, \omega\rangle \models_\lambda^B \psi$). Let $cl_{\mathbf{U}}(\psi) = \{\phi_1 \mathbf{U} \varphi_1, ..., \phi_k \mathbf{U} \varphi_k\}$ be the set of $\mathbf{U}$-formulas of $cl(\psi)$. We define $\mathcal{BP}_\psi = (P', \Gamma, \Delta', F)$ as follows: $P' = P \times 2^{cl(\psi)}$, $F = \{P \times F_{\phi_1 \mathbf{U} \varphi_1}, ..., P \times F_{\phi_k \mathbf{U} \varphi_k}\}$, where for every i, $1 \le i \le k$, $F_{\phi_i \mathbf{U} \varphi_i} = \{\Phi \subseteq cl(\psi) \mid$ if $\phi_i \mathbf{U} \varphi_i \in \Phi$ then $\varphi_i \in \Phi\}$, and Δ' is the smallest set of transition rules satisfying the following: for every $\Phi \subseteq cl(\psi)$, $p \in P, \gamma \in \Gamma$,

(α_1): if $\phi = b(x_1, ..., x_m) \in \Phi$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle(\!|p, \Phi \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \in \lambda(b(B(x_1), ..., B(x_m)))\}$;

(α_2): if $\phi = \neg b(x_1, ..., x_m) \in \Phi$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle(\!|p, \Phi \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \notin \lambda(b(B(x_1), ..., B(x_m)))\}$;

(α_3): if $\phi = \phi_1 \wedge \phi_2 \in \Phi$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_4): if $\phi = \phi_1 \vee \phi_2 \in \Phi$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_1\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_5): if $\phi = \exists x \phi_1 \in \Phi$, then:

- ($\alpha_{5.1}$): if x is not a free variable of any formula in Φ, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_x}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_1\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;
- ($\alpha_{5.2}$): otherwise, for every $c \in \mathcal{D}$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_c\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(α_6): if $\phi = \forall x \phi_1 \in \Phi$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_c \mid c \in \mathcal{D}\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(α_7): if $\phi = \phi_1 \mathbf{U} \phi_2 \in \Phi$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_1, \mathbf{X}\phi\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_8): if $\phi = \phi_1 \mathbf{R} \phi_2 \in \Phi$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p, \Phi \cup \{\phi_2, \mathbf{X}\phi\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_9): if $\Phi = \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}$ and $\langle p, \gamma\rangle \hookrightarrow \langle p', \omega\rangle \in \Delta$, $\langle(\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle(\!|p', \{\phi_1, ..., \phi_m\}|\!), \omega\rangle \in \Delta'$.

Intuitively, $\mathcal{BP}_\psi$ is a kind of "product" of $\mathcal{P}$ and ψ. $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \{\psi\}|\!), B), \omega\rangle$ iff $\mathcal{P}$ has an execution π starting from $\langle p, \omega\rangle$ s.t. π satisfies ψ under B. The control locations of $\mathcal{BP}_\psi$ are of the form $(\!|p, \Phi|\!)$, where Φ is a set of formulas, because the satisfiability of a single formula ϕ may depend on several other formulas. E.g., the satisfiability of $\phi_1 \wedge \phi_2$ depends on ϕ_1 and ϕ_2. Thus, we have to store a set of formulas into the control locations of $\mathcal{BP}_\psi$. The intuition behind each rule is explained as follows. (By abuse of notation, given a set of formulas Φ, we write $\pi \models_\lambda^B \Phi$ iff $\pi \models_\lambda^B \phi$ for every $\phi \in \Phi$.) Let π be an execution of $\mathcal{P}$ from $\langle p, \omega \rangle$.

If $b(x_1, ..., x_m) \in \Phi$, then $\pi \models_\lambda^B \Phi$ iff $\pi \models_\lambda^B b(x_1, ..., x_m)$ and π satisfies all the other formulas of Φ under B. This is ensured by Item (α_1). Item (α_2) is similar to Item (α_1).

If $\phi_1 \wedge \phi_2 \in \Phi$, then, $\pi \models_\lambda^B \Phi$ iff $\pi \models_\lambda^B \phi_1$, $\pi \models_\lambda^B \phi_2$ and π satisfies all the other formulas of Φ under B. This is ensured by Item (α_3). Item (α_4) is analogous.

If $\phi_1 \mathbf{U} \phi_2 \in \Phi$, then, $\pi \models_\lambda^B \Phi$ iff $\pi \models_\lambda^B \phi_2$ holds or both ($\pi \models_\lambda^B \phi_1$ and $\pi \models_\lambda^B \mathbf{X}(\phi_1 \mathbf{U} \phi_2)$) hold, and π satisfies all the other formulas of Φ under B. This is ensured by Item (α_7). Since ϕ_2 should eventually hold, to prevent the case where the run of $\mathcal{BP}_\psi$ always carries ϕ_1 and $\mathbf{X}(\phi_1 \mathbf{U} \phi_2)$ and never ϕ_2, we set $P \times F_{\phi_1 \mathbf{U} \phi_2} = P \times \{\Phi' \subseteq cl(\psi) \mid \text{if } \phi_1 \mathbf{U} \phi_2 \in \Phi' \text{ then } \phi_2 \in \Phi'\}$ as a set of accepting control locations. Then, the accepting run of $\mathcal{BP}_\psi$ will infinitely often visit some control locations in $P \times F_{\phi_1 \mathbf{U} \phi_2}$ which guarantees that ϕ_2 eventually holds. Item (α_8) is similar to Item (α_7).

If $\Phi = \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}$, then $\pi \models_\lambda^B \Phi$ iff $\pi^1 \models_\lambda^B \{\phi_1, ..., \phi_m\}$. It is ensured by Item (α_9).

If $\forall x \phi \in \Phi$, then $\pi \models_\lambda^B \Phi$ iff $\pi \models_\lambda^B \forall x \phi$ and $\pi \models_\lambda^B \Phi \setminus \{\forall x \phi\}$. Since $\pi \models_\lambda^B \forall x \phi$ iff $\pi \models_\lambda^B \bigwedge_{c \in \mathcal{D}} \phi_c$, where ϕ_c is ϕ_1 where x is substituted by c, we replace $\forall x \phi$ by $\bigwedge_{c \in \mathcal{D}} \phi_c$. This is expressed by Item (α_6).

If $\exists x \phi \in \Phi$, then the construction depends on whether x is a free variable of some formula in Φ or not:

- if x is not a free variable of any formula in Φ, then $\pi \models_\lambda^B \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models_\lambda^{B[x \leftarrow c]} \phi$ and $\pi \models_\lambda^B \Phi \setminus \{\exists x \phi\}$. Since x is not a free variable of any formula in Φ, we can get that $\pi \models_\lambda^B \Phi \setminus \{\exists x \phi\}$ iff $\pi \models_\lambda^{B[x \leftarrow c]} \Phi \setminus \{\exists x \phi\}$ for every $c \in \mathcal{D}$. This implies that $\pi \models_\lambda^B \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models_\lambda^{B[x \leftarrow c]} \phi$ and $\pi \models_\lambda^{B[x \leftarrow c]} \Phi \setminus \{\exists x \phi\}$. This is ensured by Item ($\alpha_{5.1}$).
- otherwise, if x is a free variable of some formula φ in Φ, we cannot apply Item ($\alpha_{5.1}$). Indeed, it may happen that ϕ is satisfied only when $x = c$, φ is not satisfied when $x = c$, whereas $\pi \models_\lambda^B \{\varphi, \exists x \phi\}$. In this case, we apply Item ($\alpha_{5.2}$). Since $\pi \models_\lambda^B \Phi$ iff $\pi \models_\lambda^B \bigvee_{c \in \mathcal{D}} \phi_c$ and $\pi \models_\lambda^B \Phi \setminus \{\exists x \phi\}$, where ϕ_c is ϕ where x is substituted by c. Since $\pi \models_\lambda^B \bigvee_{c \in \mathcal{D}} \phi_c$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models_\lambda^B \phi_c$, then, $\pi \models_\lambda^B \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models_\lambda^B \phi_c$ and $\pi \models_\lambda^B \Phi \setminus \{\exists x \phi\}$. This is ensured by Item ($\alpha_{5.2}$). Note that we can use Item ($\alpha_{5.2}$) even in the previous case when x is not a free variable of any formula in Φ. However, it is more efficient to use Item ($\alpha_{5.1}$) in this case, since Item ($\alpha_{5.1}$) adds only one symbolic transition rule, whereas Item ($\alpha_{5.2}$) adds $|\mathcal{D}|$ symbolic transition rules.

Thus, we can show that:

Theorem 2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : \mathrm{AP}_\mathcal{D} \to 2^P$, and a LTPL formula ψ, we can construct a GSBPDS $\mathcal{BP}_\psi$ with $\mathrm{O}((|\Delta| + |P| \cdot |\Gamma|) \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$ transition rules and $\mathrm{O}(|P| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$ states s.t. for every $B \in \mathcal{B}$ and every configuration $\langle p, \omega \rangle \in P \times \Gamma^*$, $\langle p, \omega \rangle$ satisfies ψ under B iff $\langle ((\!|p, \{\psi\}|\!), B), \omega \rangle \in L(\mathcal{BP}_\psi)$.*

Note that we do not need to consider all the possible subsets of $cl(\psi)$ during the construction of $\mathcal{BP}_\psi$. In order to get the above complexity, we can maintain a set of sets of formulas which are reachable from the configuration carrying the set $\{\psi\}$.

From Proposition 2, Theorem 2 and Theorem 1, we have:

Theorem 3. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : \mathrm{AP}_{\mathcal{D}} \to 2^P$ and a LTPL formula ψ, for every $B \in \mathcal{B}$ and configuration $\langle p, \omega \rangle$, whether $\langle p, \omega \rangle$ satisfies ψ under B or not can be decided in time* $\mathrm{O}(|cl_{\mathrm{U}}(\psi)| \cdot |P| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot (|\Delta| + |P| \cdot |\Gamma|)^2 \cdot 2^{3|\psi|} \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.

The complexity follows from the fact that the number of transition rules (resp. states) of the SBPDS equivalent to $\mathcal{BP}_\psi$ is at most $O(|cl_{\mathrm{U}}(\psi)| \cdot (|\Delta| + |P| \cdot |\Gamma|) \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$ (resp. $O(|cl_{\mathrm{U}}(\psi)| \cdot |P| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$), and the environments B only need to consider the variables that are used in ψ.

Remark 1. To do LTPL model-checking for PDSs, by Proposition 1, we can translate LTPL formulas into LTL formulas and apply LTL model-checking for PDSs [6,10]. This can be done in time $O(2^{3|\psi| \cdot |\mathcal{D}|^{(|cl_\forall(\psi)| + |cl_\exists(\psi)|)}})$. Our approach has a better complexity.

5 SLTPL Model-Checking for PDSs

In this section, we show how to do SLTPL model-checking for PDSs. We follow the approach of [9]. Fix a PDS $\mathcal{P}$, a set of variables $\mathcal{X}$ over $\mathcal{D}$ and a SLTPL formula ψ. Roughly speaking, for each RVE e of ψ, we construct a kind of finite automaton $\mathcal{V}$ recognizing all the configurations $(\langle p, \omega \rangle, B) \in P \times \Gamma^* \times \mathcal{B}$ s.t. $(\langle p, \omega \rangle, B) \in L(e)$. Then, we compute a SPDS $\mathcal{P}'$ which is a kind of synchronization of $\mathcal{P}$ and the $\mathcal{V}s$ that allows to determine whether the stack predicates hold at a given step by looking only at the top of the stack of $\mathcal{P}'$. Having $\mathcal{P}'$ allows to readapt the construction of Section 4 and reduce the SLTPL model-checking problem for PDSs to the emptiness problem of SBPDSs.

5.1 Extended Finite Automata

To represent RVEs, we introduce extended finite automata, in which transition rules can be labeled by a set of variables and/or their negations. Formally, let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS and $\xi = \{\alpha, \neg\alpha \mid \alpha \in \Gamma \cup \mathcal{X}\}$, an *Extended Finite Automaton* (EFA) $\mathcal{V}$ is a tuple $(\mathcal{S}, \Lambda, \Gamma, s_0, S_f)$ where $\mathcal{S}$ is a finite set of states, Γ is the input alphabet, $s_0 \in \mathcal{S}$ is the initial state, $S_f \subseteq \mathcal{S}$ is a finite set of final states, and Λ is a finite set of transition rules of the form $s_1 \overset{\ell}{\mapsto} s_2$ s.t. $s_1, s_2 \in \mathcal{S}$, $\ell \subseteq \xi$. Let $B \in \mathcal{B}$ be an environment, $\gamma \in \Gamma$ the input letter, suppose $\mathcal{V}$ is at state s_1 and $t = s_1 \overset{\ell}{\mapsto} s_2$ is a transition rule in Λ, then $\mathcal{V}$ can move to the state s_2 (i.e., s_2 is an immediate successor of s_1 under B over γ), denoted by $s_1 \overset{\gamma}{\twoheadrightarrow}_B s_2$, iff the following conditions hold: (1) for every $\alpha \in \ell$, $B(\alpha) = \gamma$; (2) for every $\neg\alpha \in \ell$, $B(\alpha) \neq \gamma$ (note that $B(\gamma) = \gamma$ if $\gamma \in \Gamma$). Obviously, the transition t will never be fired when either $\gamma_1, \gamma_2 \in \ell \cap \Gamma$ s.t. $\gamma_1 \neq \gamma_2$ or $\alpha, \neg\alpha \in \ell$ for some $\alpha \in \Gamma \cup \mathcal{X}$. This implies that ℓ can contain only one letter from Γ, and for each $\alpha \in \mathcal{X} \cup \Gamma$, ℓ cannot contain both α and $\neg\alpha$. $\mathcal{V}$ recognizes (accepts) a word $\gamma_0 ... \gamma_n$ over Γ under B iff $\mathcal{V}$ has a run $s_0 \overset{\gamma_0}{\twoheadrightarrow}_B s_1 ... s_n \overset{\gamma_n}{\twoheadrightarrow}_B s_{n+1}$ s.t. $s_{n+1} \in S_f$. Let $L(\mathcal{V})$ be the set of all the configurations $(\langle p, \omega \rangle, B) \in P \times \Gamma^* \times \mathcal{B}$ s.t. $\mathcal{V}$ recognizes ω under B.

A EFA $\mathcal{V}$ is *deterministic* (resp. *total*) iff for every state $s \in \mathcal{S}$, environment $B \in \mathcal{B}$, letter $\gamma \in \Gamma$, s has *at most* (resp. *at least*) one immediate successor $s' \in \mathcal{S}$ under B over γ. We can show that:

Proposition 3. *For every EFA $\mathcal{V} = (\mathcal{S}, \Lambda, \Gamma, s_0, S_f)$, we can compute in time* $O(2^{|\Lambda|})$ *a deterministic and total EFA $\mathcal{V}'$ s.t. $L(\mathcal{V}) = L(\mathcal{V}')$.*

Theorem 4. *For every regular predicate $e \in \mathcal{R}$, we can get in polynomial time an EFA $\mathcal{V}_e$ s.t. $L(e) = L(\mathcal{V}_e)$.*

Given a configuration $(\langle p, \gamma_1 ... \gamma_m \rangle, B) \in P \times \Gamma^* \times \mathcal{B}$, its *reverse* $(\langle p, \gamma_1 ... \gamma_m \rangle, B)^r$ is the configuration $(\langle p, \gamma_m ... \gamma_1 \rangle, B)$. Given a set $L \subseteq P \times \Gamma^* \times \mathcal{B}$, its reverse L^r is the set $\{(\langle p, \gamma_m ... \gamma_1 \rangle, B) \mid (\langle p, \gamma_1 ... \gamma_m \rangle, B) \in L\}$. We can show that:

Proposition 4. *For every EFA $\mathcal{V}$, we can get an EFA $\mathcal{V}^r$ in linear time s.t. $L(\mathcal{V})^r = L(\mathcal{V}^r)$.*

Remark 2. To represent RVEs, [20] uses automata with alternating transition rules, called Variable Automata (VA). If we use VAs to represent variable expressions, we will obtain an alternating SBPDS when synchronizing the SLTPL formula with the PDS. We introduce EFAs to avoid using alternation, since checking the emptiness of alternating SBPDSs is exponential in the size of the PDSs [20]. [11] introduces another kind of VAs, which is not suitable for our purpose, since determinizing a VA as defined in [11] is undecidable, but, we need the automata to be deterministic as will be explained later.

5.2 Storing States into the Stack

We fix a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a SLTPL formula ψ. Let $\{e_1, ..., e_n\}$ be the set of RVEs used in ψ. We suppose w.l.o.g. that $\mathcal{P}$ has a bottom-of-stack $\bot \in \Gamma$ that is never popped from the stack. For every i, $1 \le i \le n$, let $\mathcal{V}^i = (\mathcal{S}^i, \Lambda^i, \Gamma, s_0^i, S_f^i)$ be a deterministic and total EFA s.t. $L(e_i)^r = L(\mathcal{V}^i)$. Since we have predicates over the stack, to check whether the formula ψ is satisfied, we need to know at each step which RVEs are satisfied by the stack. To this aim, we will compute a SPDS $\mathcal{P}'$ which is a kind of product of $\mathcal{P}$ and the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$, where the states of the $\mathcal{V}^i s$ are stored in the stack of $\mathcal{P}'$. Roughly speaking, the stack alphabet of $\mathcal{P}'$ is of the form $(\gamma, \overrightarrow{S})$, where $\overrightarrow{S} = \left[s_1, \cdots, s_n\right]$, $s_i \in \mathcal{S}^i$ for every i, $1 \le i \le n$, is a vector of states of the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$. For every i, $1 \le i \le n$, let $\overrightarrow{S}(i)$ denote the i^{th} component of $\overrightarrow{S}$. A configuration $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ is *consistent* iff for every i, $1 \le i \le n$, $\mathcal{V}^i$ has a run $\overrightarrow{S_0}(i) \overset{\gamma_0}{\twoheadrightarrow}_B \overrightarrow{S_1}(i) \cdots \overrightarrow{S_{m-1}}(i) \overset{\gamma_{m-1}}{\twoheadrightarrow}_B \overrightarrow{S_m}(i)$ over $\gamma_0 \cdots \gamma_{m-1}$, i.e., the reverse of the stack content $\gamma_{m-1} \cdots \gamma_0$. Intuitively, a consistent configuration $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ denotes that the stack content is $\gamma_m \cdots \gamma_0$ and the runs of the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ over $\gamma_0 \cdots \gamma_{m-1}$ reach the states $\overrightarrow{S_m}(1), ..., \overrightarrow{S_m}(n)$, respectively (note that $\gamma_0 \cdots \gamma_{m-1}$ is the reverse of the stack content $\gamma_{m-1} \cdots \gamma_0$, this is why the $\mathcal{V}^i s$ are s.t. $L(e_i)^r = L(\mathcal{V}^i)$). For every i, $1 \le i \le n$, a consistent configuration $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ satisfies e_i under the environment B iff there exists $s \in S_f^i$ s.t. $\overrightarrow{S_m}(i) \overset{\gamma_m}{\twoheadrightarrow}_B s$. I.e., whether $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ satisfies e_i under B or not depends only on the top of the stack $(\gamma_m, \overrightarrow{S_m})$.

Formally, let $\overrightarrow{\mathcal{S}} = \mathcal{S}^1 \times \cdots \times \mathcal{S}^n$ and $\overrightarrow{S_0} = \left[s_0^1, \cdots, s_0^n\right]$. We compute the SPDS $\mathcal{P}' = (P, \Gamma', \Delta')$ as follows: $\Gamma' = \Gamma \times \overrightarrow{\mathcal{S}}$ is the stack alphabet and the set Δ' of transition rules are defined as follows:

1. $\langle p_1, (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle p_2, \epsilon\rangle \in \Delta'$ iff $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, \epsilon\rangle \in \Delta$ and $\overrightarrow{S} \in \overrightarrow{\mathcal{S}}$;
2. $\langle p_1, (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle p_2, (\gamma_1, \overrightarrow{S})\rangle \in \Delta'$ iff $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, \gamma_1\rangle \in \Delta$ and $\overrightarrow{S} \in \overrightarrow{\mathcal{S}}$;
3. $\langle p_1, (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle p_2, (\gamma_2, \overrightarrow{S'})(\gamma_1, \overrightarrow{S})\rangle \in \Delta'$ iff $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_1\rangle \in \Delta$ and for every i, $1 \leq i \leq n$, $\overrightarrow{S}(i) \overset{\ell_i}{\mapsto} \overrightarrow{S'}(i) \in \Lambda^i$, where $\Theta = \{(B, B) \mid B \in \mathcal{B}, \forall i : 1 \leq i \leq n, (x \in \ell_i \Longrightarrow B(x) = \gamma_1) \wedge (\neg y \in \ell_i \Longrightarrow B(y) \neq \gamma_1)\}$.

Intuitively, the run of $\mathcal{P}$ reaches the configuration $\langle p_1, \gamma_m, \cdots \gamma_0\rangle$ and the runs of the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ over the stack word $\gamma_0 \cdots \gamma_{m-1}$ reach the states $\overrightarrow{S_m}(1), ..., \overrightarrow{S_m}(n)$, respectively, iff the run of $\mathcal{P}'$ reaches the consistent configuration $\langle (p_1, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$. If $\mathcal{P}$ moves from $\langle p_1, \gamma_m, \cdots \gamma_0\rangle$ to $\langle p_2, \gamma_{m-1}, \cdots \gamma_0\rangle$ using the rule $\langle p_1, \gamma_m\rangle \hookrightarrow \langle p_2, \epsilon\rangle$, then the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ should be at $\overrightarrow{S_{m-1}}(1), ..., \overrightarrow{S_{m-1}}(n)$ after reading the stack word $\gamma_0 \cdots \gamma_{m-2}$, i.e., $\mathcal{P}'$ moves from $\langle (p_1, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$ to $\langle (p_2, B), (\gamma_{m-1}, \overrightarrow{S_{m-1}}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$. This is ensured by Item 1. The intuition behind Item 2 is similar.

If $\mathcal{P}$ moves from $\langle p_1, \gamma'_m\gamma_{m-1} \cdots \gamma_0\rangle$ to $\langle p_2, \gamma_{m+1}\gamma_m \cdots \gamma_0\rangle$ using the rule $\langle p_1, \gamma'_m\rangle \hookrightarrow \langle p_2, \gamma_{m+1}\gamma_m\rangle$, then, after reading $\gamma_0 \cdots \gamma_m$, the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ should be at $\overrightarrow{S_{m+1}}(1), ..., \overrightarrow{S_{m+1}}(n)$ where for every i, $1 \leq i \leq n$, $\overrightarrow{S_m}(i) \overset{\gamma_m}{\twoheadrightarrow}_B \overrightarrow{S_{m+1}}(i)$. I.e., $\mathcal{P}'$ moves from $\langle (p_1, B), (\gamma'_m, \overrightarrow{S_m})(\gamma_{m-1}, \overrightarrow{S_{m-1}}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$ to $\langle (p_2, B), (\gamma_{m+1}, \overrightarrow{S_{m+1}})(\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$. This is ensured by Item 3. The relation $\Theta = \{(B, B) \mid B \in \mathcal{B}, \forall i : 1 \leq i \leq n, (x \in \ell_i \Longrightarrow B(x) = \gamma_m) \wedge (\neg y \in \ell_i \Longrightarrow B(y) \neq \gamma_m)\}$ in the transition rule $\langle p_1, (\gamma'_m, \overrightarrow{S_m})\rangle \overset{\Theta}{\hookrightarrow} \langle p_2, (\gamma_{m+1}, \overrightarrow{S_{m+1}})(\gamma_m, \overrightarrow{S_m})\rangle$ guarantees that for every i, $1 \leq i \leq n$, the state $\overrightarrow{S_{m+1}}(i)$ is the immediate successor of the state $\overrightarrow{S_m}(i)$ over γ_m under B in $\mathcal{V}^i$.

The fact that EFAs are deterministic guarantees that the top of the stack can infer the truth of the regular predicates. The fact that EFAs are total makes sure that the EFAs always have a successor state on an arbitrary input and environment.

5.3 Readapting the Reduction underlying Theorem 2

In this subsection, we show how to reduce the SLTPL model-checking problem for SPDSs to the emptiness problem of SBPDSs by a readaptation of the construction underlying Theorem 2. Let $\mathcal{BP}'_\psi = (P', \Gamma', \Delta'', F)$ be the GSBPDS s.t.: $P' = P \times 2^{cl(\psi)}$, $F = \{P \times F_{\phi_1 \mathbf{U} \varphi_1}, ..., P \times F_{\phi_k \mathbf{U} \varphi_k}\}$, where for every i, $1 \leq i \leq k$, $F_{\phi_i \mathbf{U} \varphi_i} = \{\Phi \subseteq cl(\psi) \mid \phi_i \mathbf{U} \varphi_i \notin \Phi$ or $\varphi_i \in \Phi\}$, and Δ'' is the smallest set of transition rules satisfying the following: for every $\Phi \subseteq cl(\psi)$, $p \in P$, $(\gamma, \overrightarrow{S}) \in \Gamma'$:

(β_1): if $\phi = b(x_1, ..., x_m) \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \in \lambda(b(B(x_1), ..., B(x_m)))\}$;

(β_2): if $\phi = \neg b(x_1, ..., x_m) \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \notin \lambda(b(B(x_1), ..., B(x_m)))\}$;

(β_3) : if $\phi = \phi_1 \wedge \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$;

(β_4): if $\phi = \phi_1 \vee \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$ and $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$;

(β_5) : if $\phi = \exists x \phi_1 \in \Phi$, then:

($\beta_{5.1}$): if x is not a free variable of any formula in Φ, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta_x}{\hookrightarrow} \langle (\!|p,\Phi\cup\{\phi_1\}\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta'$;

($\beta_{5.2}$) : otherwise for every $c\in\mathcal{D}$, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p,\Phi\cup\{\phi_c\}\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(β_6): if $\phi = \forall x\phi_1 \in \Phi$, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p,\Phi\cup\{\phi_c \mid c\in\mathcal{D}\}\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(β_7): if $\phi = \phi_1\mathbf{U}\phi_2 \in \Phi$, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p,\Phi\cup\{\phi_2\}\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta''$ and $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p,\Phi\cup\{\phi_1,\mathbf{X}\phi\}\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta''$;

(β_8): if $\phi = \phi_1\mathbf{R}\phi_2 \in \Phi$, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p,\Phi\cup\{\phi_1,\phi_2\}\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta''$ and $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p,\Phi\cup\{\phi_2,\mathbf{X}\phi\}\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta''$;

(β_9): if $\Phi = \{\mathbf{X}\phi_1,...,\mathbf{X}\phi_m\}$, for every $\langle p,(\gamma,\overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle p',\omega\rangle \in \Delta'$, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta\cap\Theta_{id}}{\hookrightarrow} \langle (\!|p',\{\phi_1,...,\phi_m\}|\!), \omega\rangle \in \Delta''$;

(β_{10}): if $\phi = e_i \in \Phi\cap\mathcal{R}$, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p,\Phi\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B,B) \mid B\in\mathcal{B}, \exists\overrightarrow{S'}, \overrightarrow{S}(i) \overset{\gamma}{\twoheadrightarrow}_B \overrightarrow{S'}(i) \wedge \overrightarrow{S'}(i)\in S_f^i\}$;

(β_{11}): if $\phi = \neg e_i \in \Phi$ s.t. $e_i\in\mathcal{R}$, $\langle (\!|p,\Phi|\!), (\gamma,\overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p,\Phi\setminus\{\phi\}|\!), (\gamma,\overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B,B) \mid B\in\mathcal{B}, \exists\overrightarrow{S'}, \overrightarrow{S}(i) \overset{\gamma}{\twoheadrightarrow}_B \overrightarrow{S'}(i) \wedge \overrightarrow{S'}(i)\notin S_f^i\}$.

The intuition behind $\mathcal{BP}'_\psi$ is similar to the one underlying Theorem 2. $\mathcal{P}$ has an execution π starting from $\langle p,\gamma_m,...,\gamma_0\rangle$ s.t. π satisfies ψ under B iff there exist states $\overrightarrow{S_m},...,\overrightarrow{S_0}$ s.t. $\langle((\!|p,\{\psi\}|\!),B),(\gamma_m,\overrightarrow{S_m})\cdots(\gamma_0,\overrightarrow{S_0})\rangle$ is consistent and $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\!|p,\{\psi\}|\!),B),(\gamma_m,\overrightarrow{S_m})\cdots(\gamma_0,\overrightarrow{S_0})\rangle$. Items ($\beta_1$), ..., ($\beta_8$) are similar to Items ($\alpha_1$), ..., ($\alpha_8$). The main differences are Items (β_9), (β_{10}) and (β_{11}).

The relation $\Theta\cap\Theta_{id}$ in Item (β_9) ensures that $\langle((\!|p',\{\phi_1,...,\phi_m\}|\!),B),\omega\omega'\rangle$ is an immediate successor of $\langle((\!|p,\{\mathbf{X}\phi_1,...,\mathbf{X}\phi_m\}|\!),B),(\gamma,\overrightarrow{S})\omega'\rangle$ in the run of $\mathcal{BP}'_\psi$ iff $\langle(p',B),\omega\omega'\rangle$ is an immediate successor of $\langle(p,B),(\gamma,\overrightarrow{S})\omega'\rangle$ in the corresponding run of $\mathcal{P}'$, as $(B,B)\in\Theta\cap\Theta_{id}$ implies that $(B,B)\in\Theta$. This implies that $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\!|p,\{\mathbf{X}\phi_1,...,\mathbf{X}\phi_m\}|\!),B),(\gamma,\overrightarrow{S})\omega'\rangle$ iff $\mathcal{P}'$ has an immediate successor $\langle(p',B),\omega\omega'\rangle$ of $\langle(p,B),(\gamma,\overrightarrow{S})\omega'\rangle$ s.t. $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\!|p',\{\phi_1,...,\phi_m\}|\!),B),\omega\omega'\rangle$.

Item (β_{10}) expresses that if $e_i\in\Phi$, then for every execution π s.t. $\pi(0) = \langle p,\gamma_m\cdots\gamma_0\rangle$, $\pi\models^B_\lambda \Phi$ iff $\pi\models^B_\lambda \Phi\setminus\{e_i\}$ and $\pi\models^B_\lambda e_i$ (i.e., $(\langle p,\gamma_m\cdots\gamma_0\rangle,B)\in L(e_i)$, meaning there exist $\overrightarrow{S_{m+1}},...,\overrightarrow{S_0}\in\overrightarrow{\mathcal{S}}$ s.t. for every j, $0\le j\le m$, $\overrightarrow{S_j}(i) \overset{\gamma_j}{\twoheadrightarrow}_B \overrightarrow{S_{j+1}}(i)$ and $\overrightarrow{S_{m+1}}(i)\in S_f^i$). This is guaranteed by Item (β_{10}) stating $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\!|p,\Phi|\!),B),(\gamma_m,\overrightarrow{S_m})\cdots(\gamma_0,\overrightarrow{S_0})\rangle$ iff $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\!|p,\Phi\setminus\{e_i\}|\!),B),(\gamma_m,\overrightarrow{S_m})\cdots(\gamma_0,\overrightarrow{S_0})\rangle$ and there exists $\overrightarrow{S_{m+1}}\in\overrightarrow{\mathcal{S}}$ s.t. $\overrightarrow{S_m}(i) \overset{\gamma_m}{\twoheadrightarrow}_B \overrightarrow{S_{m+1}}(i)$ and $\overrightarrow{S_{m+1}}(i)\in S_f^i$. The intuition behind Item (β_{11}) is similar to Item (β_{10}). We get that:

Theorem 5. *For every $(\langle p,\gamma_m\cdots\gamma_0\rangle,B)\in P\times\Gamma^*\times\mathcal{B}$, $\langle p,\gamma_m\cdots\gamma_0\rangle\models^B_\lambda\psi$ iff there exist $\overrightarrow{S_m},...,\overrightarrow{S_0}\in\overrightarrow{\mathcal{S}}$ s.t. $\langle((\!|p,\{\psi\}|\!),B),(\gamma_m,\overrightarrow{S_m})\cdots(\gamma_0,\overrightarrow{S_0})\rangle$ is consistent and is in $L(\mathcal{BP}'_\psi)$.*

From Proposition 2, Theorem 1 and Theorem 5, we obtain that:

Theorem 6. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : \mathrm{AP}_{\mathcal{D}} \rightarrow 2^P$ and a SLTPL formula ψ, for every $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$, whether or not $\langle p, \omega\rangle$ satisfies ψ under B can be decided in time* $O(|cl_{\mathrm{U}}(\psi)| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot |P| \cdot (|\Delta| + |P| \cdot |\Gamma|)^2 \cdot |\overrightarrow{\mathcal{S}}|^2 \cdot 2^{3|\psi|} \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.

6 Experiments

We implemented our techniques in a tool for malware detection. We use BDDs to compactly represent the relations Θ. We evaluated our tool on 270 malwares taken from VX Heavens and 27 benign programs taken from Microsoft Windows XP system. All the experiments were run on Fedora 13 with a 2.4GHz CPU, 2GB of memory. The time limit is fixed to 20 minutes. Moreover, we compared the performances of our techniques with SCTPL [19] and LTL with regular valuations [9] (denoted by LTLr) model-checking. Our tool was able to detect all the malwares. Due to lack of space, Tab. 1 shows some results. Time and memory are given in seconds and MB respectively. **#LOC** denotes the number of instructions of the assembly program. The result *Yes* denotes that the program is detected as a malware, otherwise the result is *No*. As can be seen in Tab. 1, in most cases, SLTPL model-checking performs better. The analysis of several malwares using SCTPL or LTLr model-checking runs out of memory or time, whereas our tool terminates and is able to detect these malwares. Moreover, using the SCTPL

Table 1. Some Results of Malware Detection

Example		#LOC	SLTPL			SCTPL			LTLr		
			Time	Memory	Result	Time	Memory	Result	Time	Memory	Result
Virus	Akez	264	13.78	59.02	Yes	14.75	15.59	Yes	**timeout**		
Virus	Alcaul.b	904	9.79	37.40	Yes	26.25	1.08	Yes	**timeout**		
Virus	Alcaul.c	347	2.05	9.40	Yes	26.52	2.45	Yes	365.53	225.67	Yes
Virus	Alcaul.d	837	0.24	0.17	Yes	23.52	20.39	Yes	**timeout**		
Email-worm	Kirbster	1261	948.52	1383.02	Yes		**o.o.m.**		**timeout**		
Email-worm	Krynos.b	18357	987.22	947.92	Yes		**o.o.m.**		**timeout**		
Email-worm	Newapt.B	11703	1120.21	1042.74	Yes		**o.o.m.**		**timeout**		
Email-worm	Newapt.F	11771	1045.17	908.35	Yes		**o.o.m.**		**timeout**		
Email-worm	Newapt.E	11717	1059.45	970.27	Yes		**o.o.m.**		**timeout**		
Email-worm	Mydoom.j	22335	89.66	40.15	Yes	200.41	48.17	Yes	**timeout**		
Email-worm	Mydoom.v	5960	10.78	19.03	Yes	66.34	16.49	Yes	1131.00	1010.24	Yes
Email-worm	Mydoom.y	26902	66.77	36.60	Yes	90.00	43.19	Yes	**timeout**		
Trojan	LdPinch.aar	1245	32.03	198.88	Yes	1.66	8.47	Yes	**timeout**		
Trojan	LdPinch.aoq	7688	46.29	234.86	Yes	7.33	10.13	Yes	**timeout**		
Trojan	LdPinch.mj	5952	39.07	199.28	Yes	5.74	8.90	Yes	**timeout**		
Trojan	LdPinch.ld	6609	8.37	13.36	Yes	5.41	4.24	Yes	452.93	410.85	Yes
Benign	Cmd.exe	35887	109.81	20.00	No		**o.o.m.**		**timeout**		
Benign	Blastcln.exe	13819	103.87	80.53	**No**	27.72	6.30	**Yes**	**timeout**		
Benign	Regsvr32.exe	1280	7.31	26.85	**No**	0.48	1.87	**Yes**	**158.06**	48.15	**No**
Benign	ipv6.exe	13700	89.14	31.04	**No**	60.45	3.14	**Yes**	**timeout**		
Benign	dplaysvr.exe	6796	35.46	30.39	**No**	17.12	2.84	**Yes**	**timeout**		
Benign	Shutdown.exe	2524	31.69	62.93	No		**o.o.m.**		**timeout**		
Benign	Regedt.exe	60	0.02	0.02	**No**	10.62	0.03	**Yes**	0.02	0.02	**No**
Benign	Java.exe	21868	184.58	27.96	**No**	78.64	238.77	**Yes**	**timeout**		

formula Ψ'_{wv} (described in Section 3.3) causes false alarms when checking 21 benign programs, whereas using SLTPL we correctly classify these programs as benign. Moreover, our tool was able to detect the well-known malware *Flame* and to detect several other malwares that could not be detected by well-known anti-viruses such as Avira, Avast, Kaspersky, McAfee, AVG, BitDefender, Eset Nod32, F-Secure, Norton, Panda, Trend Micro and Qihoo 360.

References

1. Babić, D., Reynaud, D., Song, D.: Malware Analysis with Tree Automata Inference. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 116–131. Springer, Heidelberg (2011)
2. Beaucamps, P., Gnaedig, I., Marion, J.-Y.: Behavior Abstraction in Malware Analysis. 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. 168–182. Springer, Heidelberg (2010)
3. Beaucamps, P., Gnaedig, I., Marion, J.-Y.: Abstraction-Based Malware Analysis Using Rewriting and Model Checking. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 806–823. Springer, Heidelberg (2012)
4. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. In: SREIS (2001)
5. Bonfante, G., Kaczmarek, M., Marion, J.-Y.: Architecture of a Morphological Malware Detector. Journal in Computer Virology 5, 263–270 (2009)
6. 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)
7. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th USENIX Security Symposium (2003)
8. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symposium on Security and Privacy (2005)
9. Esparza, J., Kucera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. 186(2) (2003)
10. Esparza, J., Schwoon, S.: A BDD-Based Model Checker for Recursive Programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
11. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable Automata over Infinite Alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010)
12. Hodkinson, I., Wolter, F., Zakharyaschev, M.: Monodic Fragments of First-Order Temporal Logics: 2000-2001 A.D. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 1–23. Springer, Heidelberg (2001)
13. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting Malicious Code by Model Checking. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
14. Kupferman, O., Piterman, N., Vardi, M.Y.: An Automata-Theoretic Approach to Infinite-State Systems. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 202–259. Springer, Heidelberg (2010)
15. Lakhotia, A., Boccardo, D.R., Singh, A., Manacero, A.: Context-sensitive analysis of obfuscated x86 executables. In: PEPM (2010)

16. Lakhotia, A., Kumar, E.U., Venable, M.: A method for detecting obfuscated calls in malicious binaries. IEEE Trans. Software Eng. 31(11) (2005)
17. Singh, P.K., Lakhotia, A.: Static verification of worm and virus behavior in binary executables using model checking. In: IAW (2003)
18. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for büchi automata with appplications to temporal logic. Theor. Comput. Sci. 49, 217–237 (1987)
19. Song, F., Touili, T.: Efficient Malware Detection Using Model-Checking. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 418–433. Springer, Heidelberg (2012)
20. Song, F., Touili, T.: Pushdown Model Checking for Malware Detection. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 110–125. Springer, Heidelberg (2012)
21. Wang, F., Tahar, S., Mohamed, O.A.: First-Order LTL Model Checking Using MDGs. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 441–455. Springer, Heidelberg (2004)
22. Xu, Y., Cerny, E., Song, X., Corella, F., Mohamed, O.A.: Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 219–231. Springer, Heidelberg (1998)

Policy Analysis for Self-administrated Role-Based Access Control

Anna Lisa Ferrara[1], P. Madhusudan[2], and Gennaro Parlato[3]

[1] University of Bristol, UK
[2] University of Illinois, USA
[3] University of Southampton, UK

Abstract. Current techniques for security analysis of administrative role-based access control (ARBAC) policies restrict themselves to the *separate administration* assumption that essentially separates administrative roles from regular ones. The naive algorithm of tracking all users is all that is known for the analysis of ARBAC policies without separate administration, and the state space explosion that this results in precludes building effective tools. In contrast, the separate administration assumption greatly simplifies the analysis since it makes it sufficient to track only one user at a time. However, separation limits the expressiveness of the models and restricts modeling distributed administrative control. We undertake a fundamental study of analysis of ARBAC policies without the separate administration restriction, and show that analysis algorithms can be built that track only a bounded number of users, where the bound depends only on the number of administrative roles in the system. Using this fundamental insight paves the way for us to design an involved heuristic to further tame the state space explosion in practical systems. Our results are also very effective when applied on policies designed under the separate administration restriction. We implement our techniques and report on experiments conducted on several realistic case studies.

1 Introduction

Role-based access control (RBAC) has emerged in recent years as a simple and effective access control mechanism for large organizations [6, 17]. RBAC is the most popular model for large organizations [15, 1] and can implement a variety of MAC and DAC policies. RBAC is also a popular mechanism in assigning user privileges in computer systems, and is supported in several systems including Microsoft SQL Servers [2], Microsoft Active Directory (AGDLP) [3], SELinux, and Oracle DBMS. It simplifies policy specification and the management of user rights using a two tier management— it groups users into roles and assigns permissions to each role. In any organization, roles can be associated with job functions and hence role-permission assignments are relatively stable, while user-role assignments change quite frequently (e.g., personnel moving across departments, reassignment of duties, etc.). Managing the user-role permissions is significantly easier than managing user rights individually.

Administrative role-based access control (ARBAC) [16, 18] is a policy mechanism for controlling how changes can be made to the RBAC policy by various administrators. ARBAC policies are generally composed of three sub-policies: one controlling user-role assignment (URA), one controlling permission-role assignment (PRA), and one dealing with role-role assignments (RRA). A set of administrative roles is defined, users are assigned administrative roles, and a URA mechanism specifies when a user in an administrative role can grant or revoke role-assignments to users, including administrative roles as well. (PRA and RRA mechanisms are less common; we will not consider them in this paper.)

Security analysis of ARBAC systems is recognized as an extremely important problem, as an analysis tool can help designers to determine whether their policies meet required security properties. The developers of the administrative systems have an intended security goal that they want to enforce through the policy, and they set up the roles and administrative rules (formalized in ARBAC) to realize these intentions. Even though ARBAC policies are specified using simple administrative rules that intuitively meet the designers' intentions, it is often very difficult to find out subtle behaviours, yielding security breaches. This happens mainly because it is hard to foresee the whole effect of multiple administrative changes and their interaction. Security breaches include privilege escalation (e.g. an employee of a lower rank gaining access to resources meant for a higher rank), violation of separation of duty constraints that model conflict of interest (e.g., a user u simultaneously holding roles r_1 and r_2), etc.

In most cases the security analysis problem for ARBAC system can be phrased as a *role-reachability* problem: given a set of users, can any user in this set gain access to a given role `goal` using the ARBAC policy rules? Such a technical problem seems to be the most useful security question for ARBAC systems. Indeed, almost all interesting security questions can be *reduced* to the above problem (see [10, 22, 4]). Unfortunately, it is very hard to verify security of ARBAC policies precisely. The main source of complexity is that simulating the system to examine the entire set of reachable states causes an explosion in state-space, as it requires tracking *all* users' role-memberships. In a system with $|U|$ users and $|R|$ roles, this means exploring $O((2^{|R|})^{|U|})$ configurations, in the worst case! If the number of users is very small, this can be achieved in practice using model-checking techniques that use symbolic representations of state-spaces (like BDDs), but any realistic scenario would involve thousands of users, making this completely intractable. For those reasons researchers have turned to consider either restricted scenarios or abstraction techniques [22, 9, 4].

One crucial assumption that has been used to tackle the above problem is that of *separate administration* [22]. This essentially assumes that administrative roles and regular roles are disjoint so that administrative operations only affect regular roles and assignment/revocation of administrative permissions are not considered. Such a restriction greatly simplifies the analysis since is then sufficient to consider only the evolution of a *single user* (at a time) as opposed to tracking all users. On the other hand, the separate administration restriction limits the expressiveness of the model [22] and precludes the use of frameworks such as SARBAC [5], UARBAC [13] which do not assume such a limitation. In particular, the separate administration restriction is increasingly inappropriate in a world where administration is getting more and more distributed. For example, users belonging to regular roles in modern organizations are often administrators of several resources and have the right to grant administrative privileges to other users on these resources (e.g., a user may have administrative powers over access to their rooms, their files, etc., and have the ability to allow administrative access to other users for these resources). In such cases, it is important that administrative permissions may be granted and revoked. The separate administration restriction precludes the modeling of such scenarios.

Beyond Separate Administration: While much research has considered the separate administration model, the naive algorithm of tracking all users is all that is known for the security analysis of ARBAC policies without separate administration, and the state space explosion that this results in precludes building effective tools. The only relevant work here that we know of is recent work by Ferrara et al [4] that proposes using abstractions techniques; however, while this is effective in proving correct policies correct, it is not precise and in particular cannot generate attacks when the abstraction fails to prove safety.

Several basic questions of security analysis for policies without separate administration are still open: Does an analysis of these policies necessarily have to track all users? The RBAC model provides a layer of abstraction where users in a system are grouped together into roles; can this be exploited to perform an analysis that is completely independent of the number of users in the system (but dependent on the number of roles)? Finally, can security analysis for ARBAC be made scalable, given that multiple users may need to be tracked?

The goal of this paper is to answer fundamental theoretical questions on the security analysis of general ARBAC systems, and exploiting the insights gained, provide scalable tools to analyze expressive ARBAC policies.

As a **first contribution**, we show that the security analysis of ARBAC systems can be achieved completely independent of the number of users. More precisely, we show that the number of users that an analysis needs to track simultaneously depends only on the number of *administrative* roles (k), and, in fact, it is always sufficient to track $k+1$ users simultaneously. The proof of the theorem is quite non-trivial; a user u may reach a target role just using the administrators currently in the system (in which case, tracking one user would suffice). However, users in certain administrative roles may not exist and the system can evolve to drop administrative privileges of users; further the user u may *collude* with a subset of users, who could become administrators of the right kind and help the user u reach the target role. (When administrators are computer programs that automatically evolve, as is common in some scenarios, this does not even mean collusion, and is just exploitation of these software administrators). The fundamental theorem we prove shows that, however complex these collusions are, tracking $k + 1$ users always suffices.

The *proof* of the fundamental theorem leads to significant insights on the users that need to be tracked by an analysis. As a **second contribution**, we utilize such insights to build a trimming procedure that takes as input an ARBAC system, and reduces it to an often much smaller ARBAC system, that has significantly smaller sets of users, administrative roles, and rules.

The effectiveness of our procedure is amplified by an *aggressive pruning* algorithm which we propose as our **third contribution** and whose applicability is of independent interest (it is effective also for separated administration scenarios). Static pruning techniques have been previously proposed [10, 22, 4] in order to reduce the state space to explore. The basic idea behind those techniques is that of removing roles and administrative rules from the policies that are immaterial to the reachability of role `goal`.

As a **fourth contribution**, we evaluate our techniques on the policies (without separate assumption) in [19]— these systems are initially populated by thousands of users and the naive precise solution known will track all users and would fail pathetically. Our procedure significantly simplifies the policies considered, reducing the number of effective users that need to be tracked to be very small (often 3 to 4), which then leads us to solve the security problem for them. This result is the first we are aware of that shows that security analysis can be feasible without separate administration restriction.

Moreover, we evaluate our aggressive pruning technique on the realistic policies and test cases considered in [9] (with separate assumption). These policies are considered very complex to analyze given their huge number of roles and rules ranging respectively from 600 to 40k and 1k to 200k. Only under-approximate techniques have been found successful on those policies [9], which of course can find shallow errors but are not complete. We experimentally show that our pruning technique, in contrast to ones known, is extremely effective. Indeed, it reduces most of the aforementioned complex policies to equivalent systems (in terms of role-reachability) having as few roles (rules, resp.) as a single one.

Related Work. Besides the work already cited above, there are a few other works that are related to this paper. Li and Tripunitara [14] have studied the security analysis problem of ARBAC systems and identified fragments and restricted queries that can be solved in polynomial time. Sasturkar et al [20] have showed that the problem is PSPACE-complete, that most restrictions still are NP-hard, and some very restricted cases can be solved in polynomial time. Jha et al. [10] compared the use of model checking and first order logic programming for the security analysis of ARBAC and concluded that model checking is a promising approach for security analysis. Stoller et al [22] identify the *fixed-parameter complexity* of the problem, and show that the problem is tractable if we fix the number of roles. Their techniques are implemented in the RBAC-PAT tool [7]. In more recent work, Stoller et al have extended the ARBAC model to parameterized ARBAC that allows conditions that depend on parameters [21].

2 Preliminaries

Role Based Access Control. An RBAC policy is a tuple $\langle U, R, P, UA, PA, \succ\rangle$ where U, R, and P are finite sets of *users*, *roles*, and *permissions*, respectively,

$UA \subseteq U \times R$ is the *user-role assignment* relation, and $PA \subseteq P \times R$ is the *permission-role assignment* relation. A pair $(u, r) \in UA$ represents the membership of user u to role r, and $(p, r) \in PA$ means that role r has permission p. Roles are related by a *hierarchy relation* defined by the partial order $\succ$. The hierarchy relation allows to inherit permissions by one role from another in the hierarchy: for any two roles $r, r' \in R, r$ inherits all permissions of r' whenever $r \succ r'$. However, in the rest of the paper we restrict ourself to consider only RBAC policies with an empty hierarchy relation. From an analysis point of view it is always possible to transform a hierarchical RBAC system into one without hierarchy that preserves the reachability of its roles (see [20]). Furthermore, we concentrate our analysis on user-role administration. Thus, in the rest of the paper we refer to an RBAC policy as a tuple $\langle U, R, UA \rangle$.

Administrative Role Based Access Control. ARBAC policies [16, 18] describe a model for role-based administration of RBAC. They are composed of three modules: URA user-role administration, PRA permission-role administration, and RRA role-role administration. In this paper we focus on the user-role administration model which is of most practical interest. In practice user-role membership changes are the most frequent [11] when compared with changes in permission-role and role-role relationships. The URA policy describes how the user-role assignment relation UA can be modified in the evolution of the system. A central role is played by the set of administrative roles AR: the users in AR (called *administrators*) can assign and/or revoke roles to other users.

The assignment of a user to a role is subject to a *precondition* which depends only on the user's role-memberships. A precondition is a Boolean formula written as a conjunction of literals, where each literal is either in positive form r or in negative form $\neg r$, for some role r in R. To simplify the notation we represent each precondition with two subsets Pos and Neg of R. The set Pos represents the set of all roles the user must be in, as opposed to Neg which is the set of all roles which the user must not belong to.

The permission to assign users to roles is specified by a set of tuples $can\_assign \subseteq AR \times 2^R \times 2^R \times R$. The meaning of a can-assign tuple $(admin, Pos, Neg, r) \in can\_assign$ is that a member of the administrative role $admin \in AR$ can assign a user whose current role-membership satisfies the precondition (Pos, Neg) to the role $r \in R$. (In the rest of the paper we always assume that $Pos \cap Neg = \emptyset$).

Rules to remove users from roles are defined by a set $can\_revoke \subseteq AR \times R$. If $(admin, r) \in can\_revoke$ then a member of the administrative role $admin \in AR$ can revoke the membership of a user from role r (regardless the user role-membership). In the rest of the paper we refer to an URA model as a pair $\langle can\_assign, can\_revoke \rangle$.

Separate Administration Restriction. Under the separate administration restriction, the set of administrative roles will never appear as target of a can-assign or can-revoke rule. In other words, it is intrinsically assumed that administrators will never change their membership to administrative roles. We relax the separate administration restriction and allow administrative roles to be part of the set of roles R, i.e., $AR \subseteq R$.

ARBAC Systems. In this section we describe ARBAC systems as state-transition systems, and define the role-reachability problem for them. An ARBAC system is a tuple $\mathcal{S} = \langle U, R, \mathit{UA}, \mathit{can\_assign}, \mathit{can\_revoke}\rangle$ where $\langle U, R, \mathit{UA}\rangle$ is an RBAC policy and $\langle \mathit{can\_assign}, \mathit{can\_revoke}\rangle$ is an URA model over the set of roles R. A *configuration* of $\mathcal{S}$ is any user-role assignment relation $\mathit{UR} \subseteq U \times R$. A configuration UR is *initial* if $\mathit{UR} = \mathit{UA}$. Given two $\mathcal{S}$ configurations UR and UR', there is a *transition* from UR to UR' with rule $m \in (\mathit{can\_assign} \cup \mathit{can\_revoke})$, denoted $\mathit{UR} \xrightarrow{m} \mathit{UR}'$, if there is an *administrator ad* and an administrative role *admin* with $(\mathit{ad}, \mathit{admin}) \in \mathit{UR}$ and a user $u \in U$, and one of the following holds:
[**can-assign move**] $m = (\mathit{admin}, P, N, r)$, $P \subseteq \{t \,|\, (u,t) \in \mathit{UR}\}$, $N \subseteq R \setminus \{t \,|\, (u,t) \in \mathit{UR}\}$, and $\mathit{UR}' = \mathit{UR} \cup \{(u,r)\}$;
[**can-revoke move**] $m = (\mathit{admin}, r)$, $(u,r) \in \mathit{UR}$, and $\mathit{UR}' = \mathit{UR} \setminus \{(u,r)\}$.
A *run* of $\mathcal{S}$ is any finite sequence of $\mathcal{S}$ transitions $\pi = c_1 \xrightarrow{m_1} c_2 \xrightarrow{m_2} \ldots c_n \xrightarrow{m_n} c_{n+1}$ for some $n \geq 0$, where c_1 is an *initial* configuration of $\mathcal{S}$. An $\mathcal{S}$ configuration c is reachable if c is the last configuration of an $\mathcal{S}$ run.

Definition 1 (Role-reachability Problem). *For any role $r \in R$, r is* reachable *in $\mathcal{S}$ if there is an $\mathcal{S}$ reachable configuration UR such that $(u,r) \in \mathit{UR}$, for some $u \in U$. Given an ARBAC system $\mathcal{S}$ over the set of roles R and a role* $\mathtt{goal} \in R$, *the* role-reachability problem *asks whether* $\mathtt{goal}$ *is reachable in $\mathcal{S}$.*

3 Bounding the Number of Users to Track

In this section we show that the role-reachability problem for ARBAC is solvable by tracking at most $k+1$ users, where k is the number of administrative roles.

Theorem 1. *Let $\mathcal{S} = \langle U, R, \mathit{UA}, \mathit{can\_assign}, \mathit{can\_revoke}\rangle$ be an ARBAC system with k administrative roles. If a role $\mathtt{goal} \in \mathtt{R}$ is reachable in $\mathcal{S}$ then there exists a run of $\mathcal{S}$ in which $\mathtt{goal}$ is reachable and at most $k+1$ users change their role-combination.*

Proof. For any run $\pi = c_1 \xrightarrow{m_1} c_2 \xrightarrow{m_2} \ldots c_n \xrightarrow{m_n} c_{n+1}$ of $\mathcal{S}$, we denote with $\rho(\pi) = \mathit{admin}_1, \mathit{admin}_2, \ldots, \mathit{admin}_n$ the sequence of administrative roles where admin_j is the administrative role used in the j'th transition of π, i.e., for every $j \in [1,n]$, either $m_j = (\mathit{admin}_j, P_j, N_j, t_j)$ or $m_j = (\mathit{admin}_j, t_j)$. A user u is *engaged* in π iff there exists at least a transition in π that changes the role-combination of u. Moreover, u is *essential* in π if, u is engaged and for some $j \in [1,n]$, u is the only user in admin_j in c_j. We denote with $\mathit{index}_\pi(u)$ the greatest $j \in [1,n]$ such that u is the only user in role admin_j in c_j.

We now show that, for each run π of $\mathcal{S}$ in which role $\mathtt{goal}$ is reachable by a user (say *target*), it is possible to construct another run π' having at most $k+1$ engaged users. We assume that *target* reaches role $\mathtt{goal}$, for the first time, in the last configuration of π. We obtain π' from π by repeatedly applying the following rules.

Simplification rules: Let $\pi_0 = \pi$, and π_i be the run obtained after i steps.

1. If π_i contains an engaged user, but *target*, which is not essential, then pick one of them, say u, and remove from π_i all transitions changing u's role-combination.
2. If all engaged users in π_i are essential, then pick one of them, say u, such that $u \neq \mathit{target}$, and there is a transition m_j changing u's role-combination, where $j \geq \ell$ and $\ell = \mathit{index}_{\pi_i(u)}$. Then, remove from π_i all transitions changing u's role-combination after configuration c_ℓ.

Notice that the simplification process eventually terminates as we reduce the length of the run at each step. Also, each step always produces a run provided π_i is itself a run. The key observation to prove this property is that we always guarantee to leave a user in any administrative role to fire any move in π_i.

To conclude the proof, we show that any of such run π' has at most $k + 1$ engaged users. Since each engaged user u in π' (but *target*) is essential and no transition changing u's role-combination after $c_{\mathit{index}_{\pi'}(u)}$ exists, it holds that for any two distinct engaged users u_1 and u_2 in π' (both different from *target*), $\mathit{admin}_{j_1} \neq \mathit{admin}_{j_2}$, with $j_1 = \mathit{index}_{\pi'}(u_1)$ and $j_2 = \mathit{index}_{\pi'}(u_2)$. Thus, the number of engaged users in π' is at most equal to the number of administrative roles in $\mathcal{S}$ plus one (that represents user *target*). □

4 Reducing the Number of Users to Track

Theorem 1 gives an upper-bound on the number of users to track to solve the role-reachability problem. Although the number of users to engage is generally much smaller than the number of users in the system, in practice even tracking few users can be unfeasible, hence it is extremely desirable to reduce as much as possible such a parameter. From a theoretical viewpoint such a bound is tight, but it is unlikely that real world ARBAC instances incur such intricate worst case scenarios. Thus, the main objective of this section is to devise new heuristics to reduce the number of administrative roles, hence the number of users to track.

Our proposal is to provide sufficient conditions to eliminate administrative roles. An administrative role is *immaterial* if there is a user that can belong to that role forever without affecting the reachability of any other role. In particular, we first identify two criteria for an administrative role to be immaterial; then we transform the policy in such a way that immaterial administrative roles become regular ones. For this purpose, we add to the system a fresh administrative role, called *super*, and a new user whose role-membership is the sole role *super*. The first component of any rule administrated by an immaterial administrative role *admin* is replaced with *super*. This has the effect of making *admin* a regular role (i.e., a role without administrative permissions). More formally, we transform all can-assign moves $(\mathit{admin}, P, N, t) \in \mathit{can\_assign}$ into $(\mathit{super}, P, N, t)$, and similarly each can-revoke rule $(\mathit{admin}, t) \in \mathit{can\_revoke}$ into (super, t). We now present two sufficient conditions which lead to immaterial administrative roles.

The first sufficient condition for immaterial administrative role is as follows. Let *admin* be an administrative role which does not appear in negative form in any precondition, and that initially contains a user, say u. Since the removal of u from *admin* will not allow to fire more rules, we can impose that u will never be removed from *admin* making it immaterial.

For the second condition we resort to Theorem 1. If there are at least $k + 2$ users sharing the same role-membership, we can think that one of them will not be engaged in any run, making immaterial all administrative roles which those users are member of. As a side note, it is safe to keep in the system at most $k + 1$ users for each role-combination, the remaining ones which we denote as *spare* users can be eliminated.

Notice that, the more users share the same role-combination the more the second condition becomes effective. Therefore, our approach benefits from the use of any technique that may increase the number of users having the same role-combination. For instance, we can employ pruning techniques that transform a system in an equivalent one (in terms of the reachability of role `goal`) by eliminating roles and rules. Pruning techniques have been first introduced in [10] and proved (in some case) useful to reduce the state-space to analyze [10, 22, 4].

```
ReduceAdmin($\mathcal{S}$, goal)
  $\widehat{\mathcal{S}}' \leftarrow \mathcal{S}$;
  do
     $\widehat{\mathcal{S}} \leftarrow \widehat{\mathcal{S}}'$;
     $\widehat{\mathcal{S}}' \leftarrow$ Pruning($\widehat{\mathcal{S}}'$, goal);
     $\widehat{\mathcal{S}}' \leftarrow$ Immaterial($\widehat{\mathcal{S}}'$);
     $\widehat{\mathcal{S}}' \leftarrow$ Spare($\widehat{\mathcal{S}}'$);
  while ($\widehat{\mathcal{S}} \neq \widehat{\mathcal{S}}'$);
  return($\widehat{\mathcal{S}}'$);
```

Fig. 1. ReduceAdmin

Fig. 1 shows algorithm ReduceAdmin that eliminates the immaterial administrative roles. ReduceAdmin takes as input a pair ($\mathcal{S}$, `goal`), where $\mathcal{S}$ is an ARBAC system and `goal` is a role of $\mathcal{S}$ for which we want to check the reachability. ReduceAdmin returns an ARBAC systems $\widehat{\mathcal{S}}'$ that preserves the role-reachability of `goal` and reduces the number of administrative roles according to the two conditions for immaterial administrative roles given above. We assume that $\mathcal{S}$ has a special administrative role called *super*, containing a user as described above.

The algorithm recursively executes a do-while loop until no further simplification of the system is possible. At each iteration, it first executes a pruning algorithm, that preserves the reachability of `goal` aimed at reducing the number of roles and rules, and then it collapses all immaterial administrative roles into the special role *super*. Finally, spare users are eliminated from the system. It is easy to see that ReduceAdmin eventually terminates, as it shrinks the size of the system at each loop iteration.

Theorem 2 (Correctness of ReduceAdmin**).** *Let S be an ARBAC system over the set of roles R,* `goal` $\in$ `R`*, and $\mathcal{S}' =$ ReduceAdmin ($\mathcal{S}$,* `goal`*). Then, role* `goal` *is reachable in $\mathcal{S}$ iff it is reachable in $\mathcal{S}'$.*

Our experiments (see Sec. 6), instantiate *Pruning* in the algorithm of Fig. 1 with a novel pruning algorithm that we present in Sec. 5.

5 Aggressive Pruning

In this section we describe a novel pruning algorithm, called *aggressive pruning*, to eliminate roles and rules that are irrelevant to the reachability of role `goal`.

We extend previous proposals [10], [22], [4] by identifying six new pruning rules: the first three rules aim at discarding irrelevant roles while the remaining ones identify assignment/revocation rules that can be combined or eliminated.

In the rest of the section we refer to $\mathcal{S}$ as an ARBAC system with a special administrative role *super* which is never removed, and always contains a user whose sole membership is in role *super*. Intuitively, *super* subsumes all administrative roles *admin* for which we can guarantee that it always contains some user ready to perform a rule administered by *admin*. We denote as *persistent* all rules administered by *super*. At each step in the computation we refer to $\widehat{\mathcal{S}} = \langle U, R, UA, \mathit{can\_assign}, \mathit{can\_revoke}\rangle$ as the current pruned ARBAC system derived from $\mathcal{S}$. Below we introduce six pruning rules, called $\mathbf{R_i}$, for $\mathbf{i} \in \{1, \ldots, 6\}$, and denote with $[\![\widehat{\mathcal{S}}]\!]_i$ the ARBAC system resulting from the application of $\mathbf{R_i}$ to $\widehat{\mathcal{S}}$. We now formally define them and argue their correctness.

Removing Irrelevant Roles. A non-administrative role r is *irrelevant*, if each can-assign rule can be fired with any user u, regardless of u's membership to r. An irrelevant role can be eliminated from the system without affecting the reachability of `goal`. The elimination of a set of roles $X \subseteq R$ from $\widehat{\mathcal{S}}$ implies the following changes to the policy: (1) revoke all users-membership from each role in X; (2) remove all assignment/revocation rules having a role in X as target; (3) drop any role in X from each precondition of the remaining can-assign rules.

Formally, let $\mathcal{R}_{ua}(UA, X) = UA \setminus (U \times X)$; $\mathcal{R}_{ca}(\mathit{can\_assign}, X) = \{(\mathit{admin}, P \setminus X, N \setminus X, t) \mid (\mathit{admin}, P, N, t) \in \mathit{can\_assign} \wedge t \notin X)\}$; and $\mathcal{R}_{cr}(\mathit{can\_revoke}, X) = \mathit{can\_revoke} \setminus (R \times X)$. Removing the roles of X from $\widehat{\mathcal{S}}$ results into the system $\mathcal{R}(\widehat{\mathcal{S}}, X) = \langle U, R \setminus X, \mathcal{R}_{ua}(UA, X), \mathcal{R}_{ca}(\mathit{can\_assign}, X), \mathcal{R}_{cr}(\mathit{can\_revoke}, X)\rangle$.

We classify regular roles as non-negative, non-positive, and mixed. A role $r \in (R \setminus AR)$ is *non-positive* (*non-negative*, resp.), if r does not appear in positive (negative, resp.) form in the precondition of any can-assign rule; it is *mixed* if it appears both in positive and negative form in some precondition. We now identify sufficient conditions for a role (different from role `goal`) to be irrelevant in each of those categories.

Remove each regular role $r \in (R \setminus \{\texttt{goal}\})$ from $\widehat{\mathcal{S}}$ such that
$\mathbf{R_1}$: r is **non-positive** and $(\mathit{super}, r) \in \mathit{can\_revoke}$;
$\mathbf{R_2}$: r is **non-negative** and for every $(\mathit{admin}, P, N, t) \in \mathit{can\_assign}$ with $r \in P$, there is $(\mathit{admin}', P', N', r) \in \mathit{can\_assign}$ such that $P' \subseteq P \setminus \{r\}$, $N' \subseteq N \cup \{t\}$, and either $\mathit{admin}' = \mathit{admin}$ or $\mathit{admin}' = \mathit{super}$;
$\mathbf{R_3}$: r is **mixed** and both $\mathbf{R_1}$ and $\mathbf{R_2}$ hold.

Fig. 2. Sufficient conditions to remove irrelevant roles

Non-Positive Roles. A non-positive role r is irrelevant if there is a persistent can-revoke cr with target r. Indeed, since r is a non-positive role, a can-assign ca could not be fired with respect of a user u, only if r is in its precondition and u belongs to r. However, the persistent can-revoke cr can be executed before rule ca, thus enabling ca to be performed regardless of u's membership in r. We translate this property in the following pruning rule: remove the set X of all non-positive roles such that for each $r \in X$, $(super, r) \in can\_revoke$. Then, $[\![\widehat{\mathcal{S}}]\!]_1 = \mathcal{R}(\widehat{\mathcal{S}}, X)$. This pruning rule is summarised in Fig. 2 as $\mathbf{R_1}$.

Non-Negative Roles. Let r be a non-negative role. A can-assign ca could not be fired with respect to a user u, only if r is in the precondition of ca and u does not belong to r. However, if u can be assigned to r while satisfying the precondition of ca (except for u's membership to r), then the ca can be executed regardless of u's membership to r. We translate this property in the following rule: remove the set X of all non-negative roles r such that, for every $(admin, P, N, t) \in can\_assign$ with $r \in P$, there is $(admin', P', N', r) \in can\_assign$ such that $P' \subseteq P \setminus \{r\}$, $N' \subseteq N \cup \{t\}$, and either $admin' = admin$ or $admin' = super$. Then, $[\![\widehat{\mathcal{S}}]\!]_2 = \mathcal{R}(\widehat{\mathcal{S}}, X)$. This rule is summarized in Fig. 2 by $\mathbf{R_2}$.

Mixed Roles. A mixed role r is irrelevant, if it satisfies both conditions that make non-positive and non-negative roles irrelevant. $\mathbf{R_3}$ of Fig. 2 captures the removal of irrelevant mixed roles from $\widehat{\mathcal{S}}$. Then, $[\![\widehat{\mathcal{S}}]\!]_2 = \mathcal{R}(\widehat{\mathcal{S}}, X)$, where X is the set of all mixed roles of $\widehat{\mathcal{S}}$.

Removing Irrelevant Rules. We describe sufficient conditions to get rid of some assignment rules. Specifically, we partition those rules in three categories: (1) *combinable* which refer to pairs of rules that can be merged into a single one; (2) *implied* that identify pairs of rules such that one is subsumed by the other; (3) *non-fireable* corresponding to can-assign rules that cannot appear in any run.

Combinable Rules. Two can-assign rules ca_1, ca_2 can be combined into a single one if (1) they have the same target role, (2) their precondition sets are the same but a single role that appears in positive form in one can-assign rule and in negative form in the other, (3) at each time either there are administrators ready to fire both rules or none of them can be executed. Those rules can be merged in a single one where role r is removed from its precondition. Notice that this operation does not alter the set of reachable configurations since the resulting rule can be fired iff one between ca_1 and ca_2 is fireable.

Formally, let $ca_1 = (admin_1, P \cup \{r\}, N, t)$ and $ca_2 = (admin_2, P, N \cup \{r\}, t) \in can\_assign$, for some P, N. We define the predicate $Combinable(ca_1, ca_2)$ that holds true if $admin_1 = admin_2$. It is easy to see that if $Combinable(ca_1, ca_2)$ holds then ca_1 and ca_2 are combinable. Then, $[\![\mathcal{S}]\!]_4 = \langle U, R, UA, can\_assign', can\_revoke\rangle$ where $can\_assign' = (can\_assign \setminus \{ca_1 \mid \exists ca_2.\, Combinable(ca_1, ca_2)\}) \cup \{(admin, P, N, t) \mid \exists ca_1 = (admin, P \cup \{r\}, N, t), ca_2 = (admin_2, P, N \cup \{r\}, t) \in can\_assign. Combinable(ca_1, ca_2)\}$. $\mathbf{R_4}$ of Fig. 3 summarizes this rule.

Implied Rules. Consider two can-assign rules ca_1, ca_2 with the same target role. Then, ca_1 *implies* ca_2 if for every user u, whenever ca_2 is fireable on u, then also ca_1 is fireable on u. We give a sufficient condition to detect when ca_1 implies ca_2. For every pair of rules $ca_1 = (admin_1, P_1, N_1, r)$ and $ca_2 = (admin_2, P_2, N_2, r)$,

R₄: Let $ca_1 = (admin_1, P \cup \{r\}, N, t)$, $ca_2 = (admin_2, P, N \cup \{r\}, t) \in can\_assign$ for some P, N, and $admin_1 = admin_2$. Then, replace the **combinable** rules ca_1 and ca_2 with the can-assign role $ca = (admin_1, P, N, t)$.

R₅: If $ca_1 = (a_1, P_1, N_1, r)$, $ca_2 = (a_2, P_2, N_2, r) \in can\_assign$, either $admin_1 = super$ or $admin_1 = admin_2$, $P_1 \subseteq P_2$, and $N_1 \subseteq N_2$, then remove the **implied** rule ca_2 from $\mathcal{S}$.

R₆: Let $ca = (admin, P, N, t) \in can\_assign$ and let $Q = P \cap \{r \,|\, (u, r) \in UA\}$ such that $\exists i \in [1, |Q|]$ and for every $Z \subseteq Q$ with $|Z| = i$, there is no can-assign rule $(admin', P', N', r) \in can\_assign$ with $r \in Z$, $(P' \cap Q) \subseteq Z$ and $Z \cap N' = \emptyset$, then remove the **non-fireable** rule ca from $\mathcal{S}$.

Fig. 3. Sufficient conditions to remove/combine assignment rules

we define a predicate $Implies(ca_1, ca_2)$ that holds true iff the following holds: $P_1 \subseteq P_2$, $N_1 \subseteq N_2$, and either $admin_1 = super$ or $admin_1 = admin_2$. It is easy to see that if $Implies(ca_1, ca_2)$ holds, then ca_1 *implies* ca_2. Formally, $[\![\widehat{\mathcal{S}}]\!]_5 = \langle U, R, UA, can\_assign \setminus X, can\_revoke\rangle$, where $X = \{ca' \in can\_assign \,|\, \exists ca \in can\_assign.\ \ Implies(ca, ca')\}$. **R₅** of Fig. 3 captures this rule.

Non-Fireable Rules. A can-assign rule ca is *non-fireable* if for every run $\pi = c_1 \xrightarrow{m_1} \ldots c_n \xrightarrow{m_n} c_{n+1}$ of $\widehat{\mathcal{S}}$, $m_i \neq ca$ for every $i \in [1, n]$. We now give a sufficient condition that allows to detect when a $ca = (admin, P, N, t)$ is non-fireable. Let $Q = P \setminus \{r \,|\, (u, r) \in UA\}$, that is, the set of P roles that contain no member in the initial configuration of $\widehat{\mathcal{S}}$. Moreover, let $NotFireable$ be a predicate over the set of can-assign rules, such that $NotFireable(ca)$ holds true iff the following holds: there exists $i \in [1, |Q|]$ such that for every $Z \subseteq Q$ with $|Z| = i$, there is no rule $(a', P', N', r) \in can\_assign$ with $r \in Z$, $(P' \cap Q) \subseteq Z$ and $Z \cap N' = \emptyset$.

The following lemma holds:

Lemma 1. *Let $ca \in can\_assign$. If $NotFireable(ca)$ holds true, then can-assign rule ca is non-fireable.*

Formally, $[\![\widehat{\mathcal{S}}]\!]_6 = \langle U, R, UA, can\_assign \setminus NF, can\_revoke\rangle$, where NF is the set of all ca such that $NotFireable(ca)$ holds true. **R₆** of Fig. 3 captures this rule.

The correctness of all pruning rules is summarized by the following lemma:

Lemma 2. *Let $\widehat{\mathcal{S}}$ be an ARBAC system. For every $i \in \{1, \ldots, 6\}$, (1) $[\![\widehat{\mathcal{S}}]\!]_i$ is an ARBAC system, (2)* `goal` *is reachable in $\widehat{\mathcal{S}}$ iff* `goal` *is reachable in $[\![\widehat{\mathcal{S}}]\!]_i$, and (3) $|[\![\widehat{\mathcal{S}}]\!]_i| < |\widehat{\mathcal{S}}|$ or $|[\![\widehat{\mathcal{S}}]\!]_i| = |\widehat{\mathcal{S}}|$.*

AGGRESSIVEPRUNING **Algorithm.** takes as input an ARBAC system $\mathcal{S}$ and a role `goal` of $\mathcal{S}$ and returns a system $\mathcal{S}'$ obtained by repeatedly applying the pruning rules $\mathbf{R_i}$, for $\mathbf{i} \in \{1, \ldots, 6\}$, along with some existing pruning rules described in [4, 22]. The algorithm eventually terminates as the application of each pruning rule reduces or leaves unaltered the ARBAC system.

Theorem 3. *Let $\mathcal{S}$ be an ARBAC system and $\mathcal{S}' = AggressivePruning(\mathcal{S},$ `goal`$)$. Role* `goal` *is reachable in $\mathcal{S}'$ iff* `goal` *is reachable in $\mathcal{S}$.*

6 Experimental Results

We have implemented the procedure ReduceAdmin of Sec. 4 along with AggressivePruning of Sec. 5. Here, we evaluate both procedures on several benchmarks from the literature. The experiments are conducted on a Macbook Pro with an Intel Core i5 2.3 GHz processor and 4GB of RAM. The results of our experiments are reported in Tables 1-3. The tables report the number of roles and rules for each original ARBAC policy. Table 3, in addition reports also the number of administrative roles and the number of users for each policy. All tables report the same information about the original policies, about the policies obtained after the application of our procedures, as well as the time taken.

AggressivePruning **Evaluation on a Bank Policy.** The first set of experiments are conducted on a policy modeling a bank [9]. The bank comprises several branches, each consisting of four divisions with five non-managerial roles and two managerial ones. The policy is designed in a way that in any branch a user (non-manager) can be part of at most three non-managerial roles out of five (see [8] for details). The policy has 612 roles and 6142 rules.

Table 1. AggressivePruning on the Bank Policy

	ARBAC Policy		**After Pruning**		
	#roles	#rules	#roles	#rules	Time
1	612	6142	0	0	3.0s
	612	6142	2	1	3.0s
	612	6142	2	1	3.0s
2	612	6142	0	0	2.0s
	612	6142	0	0	2.0s
	612	6142	2	1	2.0s
3	612	6142	468	3285	0.0s
4	612	6142	462	3272	0.1s

The evaluation of our pruning procedure on the bank policy is shown in Table 1 that is divided in four sets of experiments, one for a different security query on the policy. The first query is: *Can any user (non-manager) be assigned to four non-managerial roles in a business division in any of the branches?* The first experiment considers the original policy. After a single iteration the pruning algorithm eliminates some combinable and implied rules and finds that rules assigning users to the role `goal` are non-fireable. For the remaining two experiments, we introduce errors in the policy: we add can-assign rules that allow a (non-manager) user to be part of four non-managerial roles in one of the divisions, respectively, in a single branch (second experiment) and in all branches (third experiment). In both cases, after some iterations that involve all six pruning rules, the simplified system is left with the sole role `goal`, and a single can-assign rule: $(admin, \emptyset, \emptyset, \texttt{goal})$ witnessing that `goal` is reachable.

For the second set of experiments the query is: *Can any user (non-manager) be assigned to four non-managerial roles in a business division in all the branches?* As above, the first experiment is done on the original policy, while the other two experiments on the modified policies. In the first and second experiment the pruning algorithm finds out that `goal` is unreachable (rules assigning users to `goal` are non-fireable), while in the third experiment it returns a system constituted by the sole role `goal` and a single can-assign rule: $(admin, \emptyset, \emptyset, \texttt{goal})$.

Table 2. AGGRESSIVEPRUNING on Complex Policies with Separate Administration

Size Policy		After Aggressive Pruning								
		First Suite			Second Suite			Third Suite		
#roles	#rules	#roles	#rules	Time	#roles	#rules	Time	#roles	#rules	Time
3	15	1	1	0.0s	3	5	0.0s	3	5	0.0s
5	25	1	1	0.0s	1	1	0.0s	1	1	0.0s
20	100	1	1	0.0s	11	26	0.0s	11	26	0.0s
40	200	1	1	0.0s	1	1	0.0s	1	1	0.1s
200	1000	1	1	0.1s	1	1	0.1s	1	1	0.1s
500	2500	1	1	0.1s	1	1	0.1s	1	1	0.2s
4000	20000	1	1	6.0s	1	1	6.0s	1	1	6.3s
20000	80000	1	1	3m24s	1	1	3m32s	1	1	3m20s
30000	120000	1	1	8m14s	1	1	8m33s	1	1	7m47s
40000	200000	1	1	14m50s	1	1	18m7s	1	1	21m1s

The single experiment in the third set considers the query: *Can any user (manager included) retain permissions of four non-managerial roles in a business division in any of the branches?* While the query in the forth set is: *Can any user (manager included) retain permissions of four non-managerial roles in a business division in all of the branches?* Both queries are carried out on the original policy and in both cases the pruning algorithm makes use of all six rules, reducing the system approximatively by a third.

AGGRESSIVEPRUNING **Evaluation on Big Policies.** The authors of [9] created three sets of complex test suites capturing the complexity of realistic systems. Each suite comprises ten policies where the number of roles and rules ranges respectively from 3 to 40k and 15 to 200k. Each suite presents different kind of source of complexity. Sources of complexity are parameters such as the size of the policy and type of administrative rules. We refer to [9] for a comprehensive description. We evaluate our pruning on these policies, whose results are summarized in Table 2. In [9] it has been experimentally proven that existing static pruning techniques [10, 22] are ineffective on such complex policies. In contrast, our aggressive pruning is extremely effective making the policies orders of magnitude smaller. In these policies $\mathbf{R_5}$ (implied rules) plays an essential role.

REDUCEADMIN **Evaluation.** We evaluate REDUCEADMIN on two sets of realistic ARBAC policies without separate administration, used in several case studies [22, 7, 9, 21]: a hospital and a university policy [19]. Table 3 summarizes the results of our evaluation. Besides the information of the original and the simplified policy, we also indicate whether the role `goal` is reachable.

The first set of experiments concerns the hospital policy. The first experiment in the table tests that a user is not a member of both the roles *Receptionist* and *Doctor*. The second one is meant to check that a patient is not his own primary doctor. The third experiment checks that *nurse* and *doctor* roles are disjoint. The last experiment tests if a *doctor* is able to assign a user to the role that groups patients with third party consent.

Table 3. ReduceAdmin evaluation

	ARBAC Policy					**After** ReduceAdmin					Reach
	name	#roles	#rules	#admin	#users	#roles	#rules	#admin	#users	Time	
1	Hospital	12	25	6	1092	3	5	2	9	0.3s	NO
	Hospital	12	25	6	1092	5	9	3	19	0.0s	NO
	Hospital	12	25	6	1092	3	5	2	9	0.0s	YES
	Hospital	12	25	6	1092	6	8	3	16	0.0s	YES
2	University	32	130	9	943	3	2	1	1	0.2s	NO
	University	32	130	9	943	1	1	1	1	0.2s	YES
	University	32	130	9	943	12	16	2	23	0.2s	NO
	University	32	130	9	943	11	13	2	21	0.2s	YES
	University	32	130	9	943	14	25	3	34	0.2s	YES

For the University policy, the first experiment verifies whether a user can be an explicit member of both roles *DeptChair* and *Dean*. The second experiment checks if a user may have both privileges of *Dean* and *DeptChair*. The third experiment tests if a user can be simultaneously an explicit member of both *Undergrad* and *Grad* roles. The last experiment verifies that a user can be simultaneously in the roles *GradAdmissionsCom* and *AdmissionsOfficer*.

Observations: Our techniques allow to significantly reduce, not only the number of roles and rules, but also the number of administrative roles and the amount of distinct user role-combinations. For instance, in the hospital policy, we need to consider at most 19 users out of the initial 1092, each with at most 6 roles, and only 2 or 3 of them need to be tracked! These two experiments, particularly benefit from the application of the pruning rule $\mathbf{R_5}$ (implied rules). The pruning rules concerning the removal of positive and negative roles play a significant role for the fourth university experiment and the third and the fourth hospital experiments. Such policy simplifications allowed us to fully check the role-reachability problem for such policies by using off-the-shelf tools. For example, we encoded the reduced policies into Boolean programs and then used Getafix [12] that fully verified them in few seconds. We did the same exercise with the original policies and tried several model-checking tools for finite state systems and none of them could terminate the analysis.

7 Conclusions

We have laid out the foundations of reasoning with ARBAC policies where users self-administer the resources, without recourse to a separate set of administrators. We have identified a small model theorem for analysis of such policies, arguing that tracking a bounded number of users suffices to check the policy. Using this technical insight, we have developed heuristics to reduce an ARBAC system and shown its effectiveness in analyzing real-world policies.

The work reported by us in [4], which presents abstraction techniques aimed at *proving* ARBAC policies correct is complementary to our work here which is a precise analysis that can find security breaches. A combination of these two approaches into a CEGAR scheme would be interesting. The broader view that we suggest is that security analysis of RBAC/ARBAC policies can be solved using model-checking and abstraction techniques commonly used in program verification. Developing such techniques for a larger range of policies beyond RBAC is an interesting future direction to pursue.

Acknowledgements. This research was partially supported by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO and NSF CCF #1018182.

References

[1] http://en.wikipedia.org/wiki/Role-based_access_control
[2] http://www.microsoft.com/sqlserver/en/us/default.aspx
[3] http://www.microsoft.com/it-it/server-cloud/windows-server/active-directory.aspx
[4] Ferrara, A.L., Madhusudan, P., Parlato, G.: Security analysis of access control policies through program verification. In: CSF, pp. 113–125. IEEE (2012)
[5] Crampton, J.: Understanding and developing role-based administrative models. In: CCS, pp. 158–167. ACM (2005)
[6] Ferraiolo, D., Kuhn, R.: Role-based access control. In: NCSC, pp. 554–563 (1992)
[7] Gofman, M.I., Luo, R., Solomon, A.C., Zhang, Y., Yang, P., Stoller, S.D.: RBAC-PAT: A Policy Analysis Tool for Role Based Access Control. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 46–49. Springer, Heidelberg (2009)
[8] Jayaraman, K., Ganesh, V., Tripunitara, M., Rinard, M.C., Chapin, S.J.: Arbac policy for a large multi-national bank (2010), http://kjayaram.mysite.syr.edu/mohawk/casestudy.pdf
[9] Jayaraman, K., Ganesh, V., Tripunitara, M.V., Rinard, M.C., Chapin, S.J.: Automatic error finding in access-control policies. In: CCS, pp. 163–174. ACM (2011)
[10] Jha, S., Li, N., Tripunitara, M.V., Wang, Q., Winsborough, W.H.: Towards formal verification of role-based access control policies. IEEE Trans. Dependable Sec. Comput. 5(4), 242–255 (2008)
[11] Kern, A.: Advanced features for enterprise-wide role-based access control. In: ACSAC, pp. 333–342. IEEE (2002)
[12] La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222. ACM (2009)
[13] Li, N., Mao, Z.: Administration in role-based access control. In: ASIACCS, pp. 127–138. ACM (2007)
[14] Li, N., Tripunitara, M.V.: Security analysis in role-based access control. In: SACMAT, pp. 126–135. ACM (2004)
[15] O'Connor, A.C., Loomis, R.J.: http://csrc.nist.gov/groups/SNS/rbac/documents/20101219_RBAC2_Final_Report.pdf
[16] Sandhu, R.S., Bhamidipati, V., Munawer, Q.: The arbac97 model for role-based administration of roles. ACM Trans. Inf. Syst. Secur. 2(1), 105–135 (1999)
[17] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer 29(2), 38–47 (1996)
[18] Sandhu, R.S., Munawer, Q.: The arbac99 model for administration of roles. In: ACSAC, pp. 229–238. IEEE (1999)

[19] Sasturkar, A., Yang, P., Stoller, S.D., Ramakrishnan, C.R.: Policy analysis for administrative role based access control. Tech. Rep., Stony Brook Univ. (2006)
[20] Sasturkar, A., Yang, P., Stoller, S.D., Ramakrishnan, C.R.: Policy analysis for administrative role based access control. In: CSFW, pp. 124–138. IEEE (2006)
[21] Stoller, S.D., Yang, P., Gofman, M.I., Ramakrishnan, C.R.: Symbolic reachability analysis for parameterized administrative role-based access control. Computers & Security 30(2-3), 148–164 (2011)
[22] Stoller, S.D., Yang, P., Ramakrishnan, C.R., Gofman, M.I.: Efficient policy analysis for administrative role based access control. In: CCS, pp. 445–455. ACM (2007)

Model Checking Agent Knowledge in Dynamic Access Control Policies

Masoud Koleini, Eike Ritter, and Mark Ryan

University of Birmingham,
Birmingham, B15 2TT, UK

Abstract. In this paper, we develop a modeling technique based on interpreted systems in order to verify temporal-epistemic properties over access control policies. This approach enables us to detect information flow vulnerabilities in dynamic policies by verifying the knowledge of the agents gained by both reading and reasoning about system information. To overcome the practical limitations of state explosion in model-checking temporal-epistemic properties, we introduce a novel abstraction and refinement technique for temporal-epistemic safety properties in ACTLK (ACTL with knowledge modality K) and a class of interesting properties that does fall in this category.

1 Introduction

Assume a conference paper review system in which all the PC members have access to the number of the papers assigned to each reviewer. Further assume that a PC member Alice can see the list of the papers that are assigned to another PC member and that are not authored by Alice. Then if Alice is the author of a submitted paper, she can find who the reviewer of her paper is by comparing the number of papers assigned to each reviewer (shown by the system) with the number of the assigned papers of that reviewer which she has access to.

The above is an example of a potential information leakage in *content management systems*, which are collaborative environments that allow users to create, store and manage data. They also allow controlling access to the data based on the user roles. In such multi-agent systems, access to the data is regulated by *dynamic access control policies*, which are a class of authorization rules that the permissions for an agent depend on the state of the system and change when agents interact with the system [1,2,3]. In complicated access control scenarios, there is always a risk that some required properties do not hold in the system. For instance and for a conference paper review system, the following properties need to hold in the policy:

- It should be impossible for the author of a paper to be assigned as the reviewer of his own paper (temporal safety property).
- There must be no way for the author of a paper to find out who is the reviewer of his paper (epistemic safety property).

Epistemic properties take *knowledge* of the agents into account. The knowledge can be gained by directly accessing the information, which complies with one of the meanings of the knowledge in ordinary language, that means the agent *sees* the truth. But agent also knows the truth when he indirectly reasons about it [4].

Information flow as a result of reasoning is a critical vulnerability in many collaborative systems like conference paper review systems, social networks and document management systems, and is difficult to detect. The complication of access control policies in multi-agent collaborative frameworks makes finding such weaknesses more difficult using non-automated mechanisms. Moreover, the state of art dynamic access control verification tools are unable to find such properties as they do not handle epistemic property verification in general. Therefore as the *first contribution* of this paper, we propose a policy authorization language and express how to use the *interpreted systems framework* [5,6] in order to model the related access control system. Using interpreted systems enables us to address misconfiguration in the policy and information disclosure to unauthorized agents by verifying temporal-epistemic properties expressed in the logic CTLK (CTL with knowledge modality K). The knowledge of an agent in our modelling covers both the knowledge gained by reasoning and by reading information when access permission is granted.

The practical limitation of interpreted systems is the state explosion for the systems of medium to large state space. There is also a limited number of research on the automated abstraction and refinement of the models defined in interpreted systems framework. As the *second contribution*, we develop an novel fully automated abstraction and refinement technique for verifying safety properties in ACTLK (which is a subset of CTLK) over an access control system modelled in the framework of interpreted systems. We extend counterexample guided abstraction refinement [7] to cover the counterexamples generated by the verification of temporal-epistemic properties and when the counterexample is tree-like [8]. In this paper, we only discuss the counterexamples with finite length paths, but this approach can be extended to the paths of infinite length using an unfolding mechanism [7]. We use a model-checker for multi-agent systems [9] and build the abstract model in its modelling language. The refinement is guided using the counterexample generated by the model-checker. The counterexample checking algorithm is provably sound and complete. We also introduce an interactive refinement for a class of epistemic properties that does not fall in ACTLK, but can specify interesting security properties.

We provide the details of the algorithms and proofs of the propositions in a technical report [10].

2 Related Work

In the area of knowledge-based policy verification, Aucher et al. [11] define *privacy policies* in terms of permitted or forbidden knowledge. The dynamic part of their logic deals with sending or broadcasting data. Their approach is limited in modeling knowledge gained by the interaction of agents in a multi-agent system. RW framework [2] has the most similar approach with ours. The transition system in RW is build over the knowledge of the active coalition of agents. In each state, the knowledge of the coalition is the accumulation of the knowledge obtained by performing actions or sampling system variables in previous transitions together with the initial knowledge. In the other words, knowledge in RW is gained by reading or altering system variables, not by reasoning about them. This is similar to PoliVer [12], which approximates knowledge by readability. Such verification tools are not able to detect information flow as a result of reasoning.

In the field of abstraction and refinement for temporal-epistemic logic, Cohen et al. [13] introduced the theory of simulation relation and existential abstraction for interpreted systems. Their approach is not automated and they have not provided how to refine the abstract model if the property does not hold and the counterexample is spurious. A recent research on abstraction and refinement for interpreted systems is done by Zhou et al. [14]. Although their work is about abstraction and refinement of interpreted systems, their paper is abstract and mainly discusses the technique to build up a tree-like counterexample when verifying ACTLK properties.

3 Interpreted Systems

Fagin et al. [6] introduced interpreted systems as the framework to model multi-agent systems in games scenarios. They introduced a detailed transition system which contains agents, local states and actions. Such a framework enables reasoning about both temporal and epistemic properties of the system.

Definition 1 (Interpreted system). *Let Φ be a set of atomic propositions and $\Omega = \{e, 1, \ldots, n\}$ be a set of agents. An* interpreted system *I is a tuple:*

$$I = \langle (L_i)_{i\in\Omega}, (P_i)_{i\in\Omega}, (ACT_i)_{i\in\Omega}, S_0, \tau, \gamma\rangle$$

where (1) L_i is the set of local states of agent i, and the set of global states is defined as $S = L_e \times L_1 \times \cdots \times L_n$. We also use the notation of L_i as the function that accepts a set of global states and returns the corresponding set of local states for agent i. For each $s \in S$, $l_i(s)$ denotes the local state of agent i in s (2) ACT_i is the set of actions that agent i can perform, and $ACT = ACT_e \times ACT_1 \times \cdots \times ACT_n$ is the set of joint actions. We also use ACT_i as the function that accepts a joint action and returns the action of agent i (3) $S_0 \subseteq S$ is the set of initial states (4) $\gamma : S \times \Phi \to \{\top, \bot\}$ is called the interpretation function *(5) $P_i : L_i \to 2^{ACT_i}\backslash\{\emptyset\}$ is the protocol for agent i which defines the set of possible actions for agent i in a specific local state (6) $\tau : ACT \times S \to S$ is called the* partial transition function *with the property that if $\tau(\alpha, s)$ is defined, then for all $i \in \Omega :\ ACT_i(\alpha) \in P_i(l_i(s))$. We also write $s_1 \xrightarrow{\alpha} s_2$ if $\tau(\alpha, s_1) = s_2$.*

Definition 2 (Reachability). *A global state $s \in S$ is* reachable *in the interpreted system I if there exists $s_0 \in S_0$, $s_1, \ldots, s_n \in S$ and $\alpha_1, \ldots, \alpha_n \in ACT$ such that for all $1 \le i \le n :\ s_i = \tau(\alpha_i, s_{i-1})$ and $s = s_n$. In this paper, we use G to denote the set of reachable states.*

For an interpreted system I and each agent i we define an epistemic accessibility relation on the global states as follows:

Definition 3 (Epistemic accessibility relation). *Let I be an interpreted system and i be an agent. We define the* Epistemic accessibility relation for agent i, *written $\sim_i$, on the global states of I by $s \sim_i s'$ iff $l_i(s) = l_i(s')$ and s and s' are reachable.*

4 CTLK Logic

We specify our properties in CTLK [15], which adds the epistemic modality K to the CTL (Computational Tree Logic). CTLK is defined as follows:

Definition 4. *Let Φ be a set of atomic propositions and Ω be a set of agents. If $p \in \Phi$ and $i \in \Omega$, then CTLK formulae are defined by:*

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid K_i\phi \mid EX\phi \mid EG\phi \mid E(\phi U \phi)$$

The symbol E is existential path quantifier and X, G and U are the standard CTL symbols. All CTLK temporal connectives including the pairs of symbols starting with universal path quantifier A can be written in terms of EX, EG and EU. For example, $AG\phi$ can be written as $\neg EF\neg\phi$. Epistemic connective K_i means "agent i knows that".

Example 1. Consider a conference paper review system. Then the safety property that says if a_2 is the reviewer of paper p_1 (the proposition $\mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2)$ is true), then a_1 does not know the fact that a_2 is the reviewer of p_1 can be written as $AG(\mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2) \rightarrow \neg K_{\mathsf{a}_1}\mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2))$. In an student information system where lecturers can assign one student as the demonstrator of another student, the property that states no two students, let us say a_2 and a_3, can be assigned as the demonstrator of each other is specified by the formula $AG(\neg(\mathsf{demOf}(\mathsf{a}_2, \mathsf{a}_3) \wedge \mathsf{demOf}(\mathsf{a}_3, \mathsf{a}_2)))$.

Definition 5 (Satisfaction relation). *For any CTLK-formula ϕ, the notation $(I, s) \models \phi$ means ϕ holds at state s in interpreted system I. The relation $\models$ is defined inductively in [16]. Given G as the set of reachable states in I, we have*

$$(I, s) \models K_i\phi \quad \Leftrightarrow \quad (I, s') \models \phi \textit{ for all } s' \in G \textit{ such that } s \sim_i s'$$

We use the notation $I \models \phi$ if for all $s_0 \in S_0$: $(I, s_0) \models \phi$.

5 Policy Syntax

Multi-agent access control systems grant or deny user access to the resources and services depending on the access rights defined in the policy. Access to the resources is divided into *write access*, which when granted, allows updating some system variables (in the context of this work, Boolean variables) and *read access*, that returns the value of some variables when granted. In this section, we present a policy syntax to define actions, permissions and evolutions. In the following section, we give semantics of the policy language by constructing an interpreted system from it.

Technical preliminaries: Let V be a finite set of variables and $Pred$ a finite set of predicates. The notation $\boldsymbol{v}$ is used to specify a sequence of distinct variables. An *atomic formula* or simply an *atom* is a predicate that is applied to a sequence of variables with the appropriate length. An access control policy is a finite set of rules defined as follows:

$$L ::= \top \mid \bot \mid w(\boldsymbol{v}) \mid L \vee L \mid L \wedge L \mid L \rightarrow L \mid \neg L \mid \forall v\ [L] \mid \exists v\ [L]$$

$$W ::= +w(\boldsymbol{v}) \mid -w(\boldsymbol{v}) \mid \forall v.\ W$$

$$W_s ::= W \mid W_s, W$$

$$A_R ::= \mathsf{id}(\boldsymbol{v}) : \{W_s\} \leftarrow L \qquad \text{Action rule}$$

$$R_R ::= \mathsf{id}(\boldsymbol{v}) : w(\boldsymbol{u}) \leftarrow L \qquad \text{Read permission rule}$$

In the above, $w \in Pred$, and $w(\boldsymbol{v})$ is an atom. L denotes a logical formula over atoms, which is the condition for performing an action or reading information. $\{W_s\}$ is the effect of the action that include the updates. $+w(\boldsymbol{v})$ in the effect means executing the action will set the value of $w(\boldsymbol{v})$ to true and $-w(\boldsymbol{v})$ means setting the value to false. In the case of $\forall v.W$ in the effect, the action updates the signed atom in W for all possible values of v. In the case that an atom appears with different signs in multiple quantifications in the effect (for instance, $w(c,d)$ in $\forall x. + w(c,x), \forall y. - w(y,d)$), then only the sign of the last quantification is considered for the atom. id indicates the identifier of the rule.

Let $a(\boldsymbol{v}) : E \leftarrow L$ be an action rule. The *free variables* of the logical formula L are denoted by $\mathbf{fv}(L)$ and are defined in the standard way. We also define $\mathbf{fv}(E) = \bigcup_{e \in E} \mathbf{fv}(e)$ where $\mathbf{fv}(\pm w(\boldsymbol{x})) = \boldsymbol{x}$ and $\mathbf{fv}(\forall x.W)$=$\mathbf{fv}(W) \backslash x$. We stipulate: $\mathbf{fv}(E) \cup \mathbf{fv}(L) \subseteq \boldsymbol{v}$. If $r(\boldsymbol{v}) : w(\boldsymbol{u}) \leftarrow L$ is a read rule, then $\mathbf{fv}(\boldsymbol{u}) \cup \mathbf{fv}(L) \subseteq \boldsymbol{v}$.

Let Σ be a finite set of objects. A *ground atom* is a variable-free atom; i.e. atoms with the variables substituted with the objects in Σ. For instance, if reviewer$\in Pred$ and Bob,Paper$\in \Sigma$, then reviewer(Bob,Paper) is a ground atom. In the context of this paper, we call the ground atoms as (atomic) *propositions*, since they only evaluate to true and false.

An *action* $\alpha : \varepsilon \leftarrow \ell$ contains an identifier α together with the *evolution rule* $\varepsilon \leftarrow \ell$, which is constructed by instantiating all the arguments in an action rule $a(\boldsymbol{v}) : E \leftarrow L$ with the objects in Σ. We refer to the whole action by its identifier α.

In an asynchronous multi-agent system, it is crucial to know the agent that performs an action. As the convention and for the rest of this paper, we consider the first argument of the action to be the agent performing that action. Therefore, in the action assignReviewer(Alice,Bob,Paper), Alice is the one that assigns Bob as the reviewer of Paper. If α is an action, then $\mathbf{Ag}(\alpha)$ denotes the agent that performs α.

A *read permission* $\rho : p \leftarrow \ell$ is constructed by substituting the arguments in read permission rule $r(\boldsymbol{v}) : w(\boldsymbol{u}) \leftarrow L$ with the objects in Σ. ρ is the identifier, p is the proposition and ℓ is the condition for reading p. As for the actions, we assume the first argument in ρ to be the agent that reads the proposition p, which is denoted by $\mathbf{Ag}(\rho)$.

Definition 6 (Policy). *An* access control policy *is a finite set of actions and read permissions derived by instantiating a set of rules with a finite set of objects.*

6 Building an Interpreted System from a Policy

In access control systems, when a read permission to a resource is granted, the resource will become a part of agent's local state which means agents knows the information. When the permission is denied, it will be removed from agent's directly accessible information. Therefore, we need to simulate this dynamic behaviour of the local states (temporary read permissions) by introducing extra variables into the model. This knowledge is called *knowledge by readability* of information. Moreover, it is a realistic approach to model access control systems in *asynchronous* manner. This is because in general and in real systems, different requests are held in a queue and processed one at a time asynchronously. An interpreted system is *asynchronous* if all joint actions contain at most one non-Λ agent action where Λ denotes no-operation.

Given a policy, we build an access control system based on interpreted systems framework by considering the requirements above. Incorporating temporary read permissions requires introducing some information into the local states. We say the proposition p is local to the agent i if its value only depends on the local state of i. In the other words, for all $s, s' \in S$ where $s \sim_i s'$ we have $\gamma(s,p) = \gamma(s',p)$.

Definition 7 (Local interpretation). *Let L_i be the set of local states of agent i in interpreted system I and Φ_i be the set of local propositions. We define the* local interpretation *for agent i as a function $\gamma_i : L_i \times \Phi_i \rightarrow \{\top, \bot\}$ such that $\gamma_i(l,p) = \gamma(s,p)$ where $l_i(s) = l$ for some global state s. We require the set of local propositions to be pairwise disjoint.*

The following lemma provides the theoretical background of modelling knowledge by readability in an interpreted system.

Lemma 1. *Let I be an interpreted system, G the set of reachable states, i an agent, Φ the set of propositions and $p \in \Phi$. Suppose that $p', p'' \in \Phi_i$. If for all $s \in G$:*

$$\text{if } \gamma_i(l_i(s), p'') = \top \text{ then } (I,s) \models p \;\Leftrightarrow\; \gamma_i(l_i(s), p') = \top$$

Then we have: $\quad \gamma_i(l_i(s), p'') = \top \quad \Rightarrow \quad (I,s) \models K_i p \vee K_i \neg p.$

We extend the interpreted systems to model knowledge by readability by incorporating all the atomic propositions that appear in the policy into the environment e. We call those propositions *policy propositions*. Now for each policy proposition p and for each agent, we introduce two local atomic propositions: p_{read} (p'' in Lemma 1) as the read permission of proposition p, and p_{loc} (p' in Lemma 1) as the local copy of p. We modify the transition function in order to satisfy the following property: for all reachable states, if p_{read} is true (agent has read access to p) in a state, then p_{loc} is assigned the same value as p. This property guarantees agent's knowledge of proposition p whenever his access to p is granted. The procedure to build the set of local propositions Φ_i and upgrading the set of actions in policy $\mathcal{C}$ into a new set $\mathcal{A}^u_{\mathcal{C}}$ which allows updating local propositions according to the Lemma 1 is presented in [10].

Symbolic transition function. Given a policy which contains a set of actions, we provide the details for calculating the symbolic transition function we use for traversing over a path in our system. Symbolic transition function applies on a set of states and returns the result of performing an action over the states of that set.

As a convention, we use $s[p \mapsto m]$ where $s \in S$ to denote the state that is like s except that it maps the proposition p to the value m. Let $st \subseteq S$ be a set of states. When performing the action $\alpha : \varepsilon \leftarrow \ell$ in the states of st, the transition is only performed in the states that satisfy the permission ℓ. In the resulting states, the propositions that do not appear in ε remain the same as in the states that the transition begins. Therefore, we define:

$$\Theta_\alpha(st) = \Big\{ s[p \mapsto \top \mid +p \in \varepsilon][p \mapsto \bot \mid -p \in \varepsilon] \;\Big|\; s \in st, (I,s) \models \ell \Big\}$$

Definition 8 (Derived interpreted system). *Let $\mathcal{C}$ be a policy with Σ_{Ag} as the set of agents, $\Phi_{\mathcal{C}}$ the set of policy propositions, Φ_i, $i \in \Sigma_{Ag}$ and $\mathcal{A}^u_{\mathcal{C}}$ the local propositions and updated set of actions in $\mathcal{C}$ constructed to modify local propositions based on Lemma [1]. Let $\Omega = \{e\} \cup \Sigma_{Ag}$ and $\Phi = \bigcup_{i\in\Omega} \Phi_i$ where $\Phi_e = \Phi_{\mathcal{C}}$. Then the interpreted system derived from policy $\mathcal{C}$ is:*

$$I_{\mathcal{C}} = \langle (L_i)_{i\in\Omega}, (P_i)_{i\in\Omega}, (ACT_i)_{i\in\Omega}, S_0, \tau, \gamma\rangle$$

where (1) L_i is the set of local states of agent i, where each local state is a valuation of the propositions in Φ_i. The set of global states is defined as $S = L_e \times L_1 \times \cdots \times L_n$ (2) $ACT_i = \{\alpha \in \mathcal{A}^u_{\mathcal{C}} \mid \boldsymbol{Ag}(\alpha) = i\} \cup \{\Lambda\}$ where Λ denotes no operation*, and a joint action is a $|\Omega|$-tuple such that at most one of the elements is non-Λ (asynchronous interpreted system). For simplicity,* we denote a joint action with its non-Λ element *(3) $S_0 \subseteq S$ is the set of initial states (4) γ is the interpretation function over S and Φ. If $p \in \Phi_i$ then we have $\gamma(s,p) = \gamma_i(l_i(s),p)$ (5) P_i is the protocol for agent i where for all $l \in L_i$: $P_i(l) = ACT_i$ (6) τ is the transition function that is defined as follows: if α is a joint action (or simply, an action) and $s \in S$, then $\tau(\alpha,s) = s'$ if $\Theta_\alpha(\{s\}) = \{s'\}$.*

The system derived from policy $\mathcal{C}$ is a special case of interpreted systems where the local states are the valuation of local propositions. In the derived model, the state of the system that is specified by policy $\mathcal{C}$ is simulated in environment e and local states store the information that are accessible to the agents.

7 Abstraction Technique

In an interpreted system, the state space exponentially increases when extra propositions are added into the system. Considering a fragment of CTLK properties known as ACTLK as the specification language, we are able to verify the properties over an over-approximated abstract model instead of the concrete one. ACTLK is defined as follows:

Definition 9. *Let Φ be the set of atomic propositions and Ω set of agents. If $p \in \Phi$ and $i \in \Omega$, then ACTLK formulae are defined by:*

$$\phi ::= p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid K_i\phi \mid AX\phi \mid A(\phi U \phi) \mid A(\phi R \phi)$$

where the symbol A is universal path quantifier which means "for all the paths".

To provide a relation between the concrete model and the abstract one, we extend the *simulation relation* introduced in [17] to cover the epistemic relation between states. Using the abstraction technique that preserves simulation relation between the concrete model and the abstract one, we are able to verify ACTLK specification formulas over the model. In this paper and for abstraction and refinement, we focus on safety properties expressed in ACLK. The advantages of safety properties are first, they are capable of expressing *policy invariants*, and second, the generated counterexample contains finite sequence of actions (or transitions). We can extend the abstraction and refinement method to the full ACTLK by unfolding the loops in the counterexamples into finite transitions as described in [7], which is outside the scope of this paper.

7.1 Existential Abstraction

The general framework of existential abstraction is first introduced by Clark et. al in [17]. Existential abstraction partitions the states of a model into clusters, or equivalence classes. The clusters form the states of the abstract model. The transitions between the clusters in the abstract model give rise to an over-approximation of the original (or concrete) model that *simulates* the original one. So, when a specification in ACTL (or in the context of this paper, ACTLK) logic is true in the over-approximated model, it will be true in the concrete one. Otherwise, a counterexample will be generated which needs to be verified over the concrete model.

Notation 1. *For simplicity, we use the same notation ($\sim_i$) for the epistemic accessibility relation in both the concrete and abstract interpreted systems.*

Definition 10 (Simulation). *Let I and $\widetilde{I}$ be two interpreted systems, Ω be the set of agents in both systems, and Φ and $\widetilde{\Phi}$ the corresponding set of propositions where $\widetilde{\Phi} \subseteq \Phi$. The relation $H \subseteq S \times \widetilde{S}$ is* simulation relation *between I and $\widetilde{I}$ if and only if:*

1. *For all $s_0 \in S_0$, there exists $\widetilde{s}_0 \in \widetilde{S_0}$ st. $(s_0, \widetilde{s}_0) \in H$.*

and for all $(s, \widetilde{s}) \in H$:

2. *For all $p \in \widetilde{\Phi}$: $\gamma(s,p) = \widetilde{\gamma}(\widetilde{s},p)$*
3. *For each state $s' \in S$ such that $\tau(s,\alpha) = s'$ for some $\alpha \in ACT$, there exists $\widetilde{s}' \in \widetilde{S}$ and $\widetilde{\alpha} \in \widetilde{ACT}$ such that $\widetilde{\tau}(\widetilde{s},\widetilde{\alpha}) = \widetilde{s}'$ and $(s', \widetilde{s}') \in H$.*
4. *For each state $s' \in S$ such that $s \sim_i s'$, there exists $\widetilde{s}' \in \widetilde{S}$ such that $\widetilde{s} \sim_i \widetilde{s}'$ and $(s', \widetilde{s}') \in H$.*

The above definition for simulation relation over the interpreted systems is very similar to the one for Kripke model [7], except that the relation for the epistemic relation is introduced. If such simulation relation exists, we say that $\widetilde{I}$ *simulates* I (denoted by $I \preceq \widetilde{I}$). If H is a function, that is, for each $s \in S$ there is a unique $\widetilde{s} \in \widetilde{S}$ such that $(s, \widetilde{s}) \in H$, we write $h(s) = \widetilde{s}$ instead of $(s, \widetilde{s}) \in H$.

Proposition 1. *For every ACTLK formula φ over propositions $\widetilde{\Phi}$, if $I \preceq \widetilde{I}$ and $\widetilde{I} \models \varphi$, then $I \models \varphi$* [14].

Variable Hiding Abstraction. Variable hiding is a popular technique in the category of existential abstraction. In our methodology, we consider factorizing the concrete state space into equivalence classes that act as abstract states by abstracting away a set of system propositions. In our approach, the states in each equivalence class are only different in the valuation of the hidden propositions. The actions in the abstract model are the equivalence classes of the actions in the concrete model. All the actions in each equivalence class have the visible propositions with the same sign in the effect of the evolution rule, and the semantically equivalent permissions when invisible propositions are existentially quantified. The abstract system simulates the concrete one (see [10] for technical details).

Definition 11. *We define $h_A : ACT \rightarrow \widetilde{ACT}$ as the surjection that maps the actions in the concrete model to the actions in the abstract one.*

8 Automated Refinement

Our counterexample based abstraction refinement method consists of three steps: (1) *Generating the initial abstraction* by building the simplest possible initial abstract model by retaining only the propositions that appear in specification φ which we aim to verify (2) *Model-checking the abstract structure*. If the abstract model satisfies φ, then it can be concluded that the concrete model also satisfies φ. If the abstract model checking generates a counterexample, it should be checked if the counterexample is an actual counterexample for the concrete model. If it is spurious, the abstract model should be refined (3) *Refining the abstraction* by partitioning the states in the abstract model in such a way that the refined model does not admit the same counterexample. For the refinement, we turn some of invisible variables into visible. After each refinement, step 2 will be proceeded.

The process of abstraction and refinement will eventually terminate, as in the worst case, the refined model becomes the same as the finite state concrete one.

8.1 Generating the Initial Abstraction

For automatic abstraction refinement, we build the initial model as simple as possible. For an ACTLK formula φ, we keep all the atomic propositions that appear in φ visible in the abstract model and hide the rest.

8.2 Validation of Counterexamples

The structure of a counterexample created by the verification of an ACTLK formula is different from the counterexample generated in the absence of knowledge modality. In an ACTLK counterexample, we have epistemic relations as well as temporal ones. Analysis of such counterexamples is more complicated than the counterexamples for temporal properties.

A counterexample for a safety property in ACTLK is a loop-free tree-like graph with states as vertices, and temporal and epistemic transitions as edges (figure 1). Every counterexample has an initial state as the root. A temporal transition in the graph is labelled with its corresponding action and epistemic transition is labelled with the corresponding epistemic relation. We define a *temporal path* as a path that contains only temporal transitions. An *epistemic path* contains at least one epistemic transition. Every state in the counterexample is *reachable from an initial state*, which may differ from the root. For any state s, we write s for the empty path which starts and finishes in s.

Counterexample Formalism: A tree is a finite set of temporal and epistemic paths with an initial state as the root. Each path begins from the root and finishes at a leaf. For an epistemic transition over a path, we use the same notation as the epistemic relation while we consider the transition to be from left to the right. For instance, the tree in the figure 1 is formally presented by $\{\widetilde{s}_0 \xrightarrow{\widetilde{\alpha}_1} \widetilde{s}_1 \xrightarrow{\widetilde{\alpha}_2} \widetilde{s}_3,\ \widetilde{s}_0 \xrightarrow{\widetilde{\alpha}_1} \widetilde{s}_1 \sim_a \widetilde{s}'_2 \xrightarrow{\widetilde{\alpha}'_3} \widetilde{s}'_3\}$.

To verify a tree-like counterexample, we traverse the tree in a *depth-first* manner. An abstract counterexample is valid in the concrete model if a real counterexample in the concrete model corresponds to it. We use the notation $s \dashrightarrow s'$ when the type of the transition from s to s' is not known.

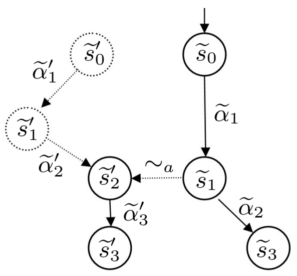


Fig. 1. A tree-like counterexample generated by the verification of an ACTLK safety property over the abstract model. In the diagram, $\widetilde{s}_0, \widetilde{s}'_0 \in S_0$ and $\widetilde{s}_1 \sim_a \widetilde{s}'_2$. As reachability is a requirement for $\widetilde{s}_1 \sim_a \widetilde{s}'_2$ and $\widetilde{s}_1$ is already reachable, the temporal path $\widetilde{s}'_0 \xrightarrow{\widetilde{\alpha}'_1} \widetilde{s}'_1 \xrightarrow{\widetilde{\alpha}'_2} \widetilde{s}'_2$ provides the witness for the reachability of $\widetilde{s}'_2$. Considering this witness is required in counterexample checking.

Definition 12 (Vertices, root). *Let $\widetilde{ce}$ be a counterexample. Then **Vert**$(\widetilde{ce})$ denotes the set of all the states that appear in $\widetilde{ce}$. **Root**$(\widetilde{ce})$ denotes the root of $\widetilde{ce}$. For a path $\widetilde{\pi}$, **Root**$(\widetilde{\pi})$ denotes the state that $\widetilde{\pi}$ starts with.*

Definition 13 (Corresponding paths). *Let $\widetilde{I}$ be an abstract model of the interpreted system I, h be the abstraction function, and h_A be the function that maps the actions in I to the ones in $\widetilde{I}$. The concrete path $\pi = s_1 \to \cdots \to s_n$ in the concrete model corresponds to the path $\widetilde{\pi} = \widetilde{s}_1 \to \cdots \to \widetilde{s}_n$ in the abstract model, if*

- *For all $1 \leq i \leq n : \widetilde{s}_i = h(s_i)$*
- *If $\widetilde{s}_i \xrightarrow{\widetilde{\alpha}_{i+1}} \widetilde{s}_{i+1}$ is a temporal transition, we have $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ where $h_A(\alpha_{i+1}) = \widetilde{\alpha}_{i+1}$.*
- *If $\widetilde{s}_i \sim_a \widetilde{s}_{i+1}$ is an epistemic transition, then $s_i \sim_a s_{i+1}$ and s_{i+1} is reachable in the concrete model.*

Definition 14 (Concrete counterexample). *Let $\widetilde{ce}$ be a tree-like counterexample in the abstract model where **Root**$(\widetilde{ce}) \in \widetilde{S}_0$. A concrete counterexample ce corresponds to $\widetilde{ce}$ if **Root**$(ce) \in S_0$ and there exists a one-to-one correspondence between the states and the paths of the counterexamples ce and $\widetilde{ce}$ according to the definition 13.*

To *verify a path* in a counterexample, we define two transition rules TEMPORALCHECK and EPISTEMICCHECK denoted by $\Rightarrow_t$ and $\Rightarrow_e$ as in figure 2. For a path with the transition $\widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}'$ as the head and for the concrete states st, the rule $\Rightarrow_t$ finds all the successors of the states in st which reside in $h^{-1}(\widetilde{s}')$. If the head of the path is the epistemic transition $\widetilde{s} \sim_a \widetilde{s}'$, then the rule $\Rightarrow_e$ extracts all the *reachable states* in $h^{-1}(\widetilde{s}')$ corresponding to π' as the witness of reachability of $\widetilde{s}'$, which has common local states with some states in $st \subseteq h^{-1}(\widetilde{s})$. Both the temporal and epistemic rules are deterministic. We write $\Rightarrow_t^*$ to denote a sequence of temporal transitions $\Rightarrow_t$. We use $\Rightarrow^*$ to denote a sequence of the transitions $\Rightarrow_t$ or $\Rightarrow_e$.

Proposition 2 (Soundness of $\Rightarrow^*$). *Let $\widetilde{\pi} = \widetilde{s}_1 \to \cdots \to \widetilde{s}_n$ be a path in the abstract model. If $st_1 \subseteq h^{-1}(\widetilde{s}_1)$ and $(\widetilde{\pi}, st_1) \Rightarrow^* (\widetilde{s}_n, st_n)$ for some $\emptyset \subset st_n \subseteq S$, then there exists a concrete path that starts from a state in st_1 and ends in a state in st_n.*

In the case that $\widetilde{\pi} = \widetilde{s}_0 \to \cdots \to \widetilde{s}_n$ is a path in the counterexample and $(\widetilde{\pi}, S_0 \cap h^{-1}(\widetilde{s}_0)) \Rightarrow^* (\widetilde{s}_n, st_n)$, then there exists a corresponding concrete path starting at some initial state $s_0 \in S_0 \cap h^{-1}(\widetilde{s}_0)$ which ends at some state $s_n \in st_n$.

$$\textsc{TemporalCheck}\ \frac{h_A^{-1}(\widetilde{\alpha}) = \{\alpha_1, \ldots, \alpha_n\}}{(\widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}' \,||\, \pi, st) \Rightarrow_t (\pi, \bigcup_{i=1}^{n} \Theta_{\alpha_i}(st) \cap h^{-1}(\widetilde{s}'))}$$

$$\textsc{EpistemicCheck}\ \frac{\begin{array}{c}\pi' = \widetilde{s}'_0 \xrightarrow{\widetilde{\alpha}'_1} \ldots \xrightarrow{\widetilde{\alpha}'_m} \widetilde{s}' \text{ is a temporal path to } \widetilde{s}' \text{ where } \widetilde{s}'_0 \in \widetilde{S}_0 \\ (\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow_t^* (\widetilde{s}', st') \qquad \hat{st} = \{s \in st' \mid l_a(s) \in L_a(st)\}\end{array}}{(\widetilde{s} \sim_a \widetilde{s}' \,||\, \pi, st) \Rightarrow_e (\pi, \hat{st})}$$

Fig. 2. Temporal and epistemic transition rules. In EPISTEMICCHECK rule, π' is the witness for the reachability of $\widetilde{s}'$ in the abstract model, and st' is the concrete states that are reachable through the concrete paths corresponding to π'. In the case that the model-checker returns all the abstract paths to $\widetilde{s}'$, let us say $\widetilde{\Pi}'$, then st' will be calculated as $st' = \bigcup\{st \mid \pi' = \widetilde{s}'_0 \to \cdots \to \widetilde{s}' \in \widetilde{\Pi}', \widetilde{s}'_0 \in \widetilde{S}_0$ and $(\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow_t^* (\widetilde{s}', st)\}$.

Proposition 3 (Completeness of $\Rightarrow^*$). *Let $\widetilde{\pi} = \widetilde{s}_1 \to \cdots \to \widetilde{s}_n$ be a path in the abstract model. If there exists a concrete path $\pi = s_1 \to \cdots \to s_n$ corresponding to $\widetilde{\pi}$ and $s_1 \in st_1 \subseteq h^{-1}(\widetilde{s}_1)$, then $(\widetilde{\pi}, st_1) \Rightarrow^* (\widetilde{s}_n, st_n)$ for some $\emptyset \subset st_n \subseteq S$.*

Forward transition rules in figure 2 are sufficient to check *linear counterexamples* or equivalently, paths. To extend the counterexample checking to tree-like counterexample, extra procedures are required.

To verify a tree-like counterexample, we introduce two transition rules BACKWARDTCHECK and BACKWARDECHECK denoted by $\Leftarrow_t$ and $\Leftarrow_e$. The transition rules find all the predecessors of the states in st (figure 3) with respect to the temporal or epistemic transitions in a backward manner which reside in the set of reachable states through the path. We write $\Leftarrow^*$ to denote a sequence of backward transitions $\Leftarrow_t$ and $\Leftarrow_e$.

Assume that $\widetilde{\pi} = \widetilde{s}_0 \to \cdots \to \widetilde{s}_n$ is a path in the counterexample $\widetilde{ce}$ which $(\widetilde{\pi}, S_0 \cap h^{-1}(\widetilde{s}_0)) \Rightarrow^* (\widetilde{s}_n, st_n)$ for some $\emptyset \subset st_n \subseteq S$. st_n contains all the states in the leaves of the concrete paths corresponding to $\widetilde{\pi}$. The point is that not all the concrete states that are traversed in $\Rightarrow^*$ can reach the states in st_n. If $\widetilde{s} \in \mathbf{Vert}(\widetilde{\pi})$, then $(\widetilde{\pi}, st_n) \Leftarrow^* (\widetilde{s}_0, st_0)$ finds the set of states $r_{\widetilde{s}}$ which contains the reachable states in $h^{-1}(\widetilde{s})$ that lead to some states in st_n along the concrete paths corresponding to $\widetilde{\pi}$. st_0 contains the initial states that lead to the states in st_n. We use the notation $r_{\widetilde{s}}^{\widetilde{\pi}}$ to relate $r_{\widetilde{s}}$ with the path $\widetilde{\pi}$. Note that to find $r_{\widetilde{s}}^{\widetilde{\pi}}$, we first need to find st_n through $\Rightarrow^*$ transition.

Assume that $\widetilde{\Pi} \subseteq \widetilde{ce}$. If $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ then we define $r_{\widetilde{s}}^{\widetilde{\Pi}} = \cap_{\widetilde{\pi} \in \widetilde{\Pi}} r_{\widetilde{s}}^{\widetilde{\pi}}$. If $\widetilde{s} \notin \mathbf{Vert}(\widetilde{\pi})$, then we stipulate $r_{\widetilde{s}}^{\widetilde{\pi}} = h^{-1}(\widetilde{s})$. We also stipulate $r_{\widetilde{s}_0}^{\emptyset} = S_0 \cap h^{-1}(\widetilde{s}_0)$ where $\widetilde{s}_0 = \mathbf{Root}(\widetilde{ce})$ and $r_{\widetilde{s}}^{\emptyset} = h^{-1}(\widetilde{s})$ for all $\widetilde{s} \in \mathbf{Vert}(\widetilde{ce})$ where $\widetilde{s} \neq \widetilde{s}_0$.

Proposition 4. *A counterexample $\widetilde{ce}$ in the abstract model has a corresponding concrete one if:*

1. *for each path $\widetilde{\pi} \in \widetilde{ce}$, there exists $\emptyset \subset st \subseteq S$ such that $(\widetilde{\pi}, S_0 \cap h^{-1}(\widetilde{s}_0)) \Rightarrow^* (\widetilde{s}', st)$ where $\widetilde{s}_0 = \boldsymbol{Root}(\widetilde{ce})$ and $\widetilde{\pi}$ ends in $\widetilde{s}'$.*
2. *for all $\widetilde{s} \in \boldsymbol{Vert}(\widetilde{ce}) : r_{\widetilde{s}}^{\widetilde{ce}} \neq \emptyset$.*

$$\textsc{BackwardTCheck}\ \frac{\begin{array}{c}(\pi, S_0 \cap h^{-1}(\mathbf{Root}(\pi))) \Rightarrow^* (\widetilde{s}, st') \\ h_A^{-1}(\widetilde{\alpha}) = \{\alpha_1, \ldots, \alpha_n\} \qquad rs = \bigcup_{i=1}^{n} \Theta_{\alpha_i}^{-1}(st) \cap st'\end{array}}{(\pi \,||\, \widetilde{s} \xrightarrow{\widetilde{\alpha}} \widetilde{s}', st) \Leftarrow_t (\pi, rs) \qquad r_{\widetilde{s}} := rs}$$

$$\textsc{BackwardECheck}\ \frac{\begin{array}{c}(\pi, S_0 \cap h^{-1}(\mathbf{Root}(\pi))) \Rightarrow^* (\widetilde{s}, st'') \\ \pi' = \widetilde{s}'_0 \xrightarrow{\widetilde{\alpha}'_1} \ldots \xrightarrow{\widetilde{\alpha}'_m} \widetilde{s}' \text{ is the temporal path to } \widetilde{s}' \text{ where } \widetilde{s}'_0 \in \widetilde{S}_0 \\ (\pi', S_0 \cap h^{-1}(\widetilde{s}'_0)) \Rightarrow^* (\widetilde{s}', st') \\ \hat{st} = \{s \in st'' \mid l_a(s) \in L_a(st \cap st')\}\end{array}}{(\pi \,||\, \widetilde{s} \sim_a \widetilde{s}', st) \Leftarrow_e (\pi, \hat{st}) \qquad r_{\widetilde{s}} := \hat{st}}$$

Fig. 3. Backward temporal and epistemic transition traversal. $\Theta_{\alpha}^{-1}(st)$ computes the set of predecessors of the states in st with respect to the transitions made by action α.

Let $\widetilde{ce}$ be a counterexample. The process of counterexample checking iterates over the paths in $\widetilde{ce}$ and checks if they corresponds to some paths in the concrete model by using proposition 2 and the transition rule $\Rightarrow^*$. If $\widetilde{\pi} \in \widetilde{ce}$ corresponds to some concrete paths, then for each state $\widetilde{s}$ in $\widetilde{\pi}$, the algorithm finds all the concrete states $r_{\widetilde{s}}^{\widetilde{\pi}}$ in $h^{-1}(\widetilde{s})$ that lead to the leaf states of the concrete paths, by applying $\Leftarrow^*$ over $\widetilde{\pi}$. In each loop iteration, the paths in $\widetilde{ce}$ that are processed in previous iterations are stored in the set $\widetilde{\Pi}$. The set $r_{\widetilde{s}}^{\widetilde{\Pi}}$ stores the concrete states that are common between the paths in $\widetilde{\Pi}$ and should remain non-empty during the process of counterexample checking. The procedure returns false if one of the paths in $\widetilde{ce}$ does not have corresponding concrete path or $r_{\widetilde{s}}^{\widetilde{\Pi}} = \emptyset$ for some $\widetilde{s} \in$ **Vert**($\widetilde{ce}$) and $\widetilde{\Pi} \subseteq \widetilde{ce}$. Otherwise it returns true [10].

8.3 Refinement of the Abstraction

For the refinement, we find the *failure state* in the counterexample as a standard terminology [7] by simulating the counterexample in the concrete model. A failure state $\widetilde{s}_f$ is a state along the tree-like counterexample where the concrete transitions cannot follow the transitions from $\widetilde{s}_f$. The concrete states that follow the counterexample and then stop following are *dead-end* states. To refine the abstract model, we split up the dead-end states from the rest of the states in $h^{-1}(\widetilde{s}_f)$ by turning a set of invisible variables into visible so that the same counterexample does not occur in the refined model by finding *conflict clauses*. See [10] for the full technical details.

8.4 Going beyond ACTLK

While this section develops a fully automated abstraction refinement method for the verification of temporal-epistemic properties that reside the category of ACTLK, some important epistemic safety properties does not fall into this category. For instance and in a conference paper review system, it is valuable for policy designers to verify that for all reachable states, an author of a paper, say a, cannot find out ($\neg K_{\mathsf{a}}$) who is the reviewer of his own paper (see the first property in example 1). Although we are able to verify such properties in the concrete model, we cannot apply automated counterexample-guided abstraction and refinement for such properties.

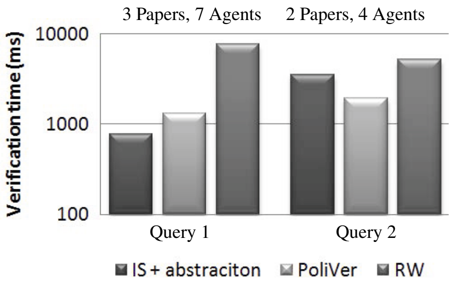
For the abstraction and refinement, we restrict the formula in scope of the knowledge operators to propositional formulas (see [10] for the technical discussion). Then we use an interactive refinement procedure in the following way: we abstract the interpreted system in the standard way that we described. If the property does not hold in the abstract model, the counterexample will be checked in the concrete model and the abstract model will be refined if it is required. If the property turned to be true in the abstract model as a result of the satisfaction of $\neg K_{\mathsf{a}}$ (for which there is no witness in the abstract model), then we refine the local state of the agent a in an interactive manner. In this way, the tool asks the user to selects a set of invisible *local propositions*, with possibly higher correlation to the agent's knowledge, to be added in the next round if required. This process will continue until a valid counterexample appear while the local state is still abstract, or the local state becomes concretized.

9 Experimental Results

We have implemented a tool in F# functional programming language. The front end is a parser that accepts a set of action and read permission rules, a set of objects and a query in the form of $\iota : \varphi$ where ι is the formula representing the initial states and φ is the property we aim to verify. Given the above information, the tool derives an interpreted system based on definition 8 where the initial states of the system are determined by parameter ι in the query. On the back end, we use MCMAS [9] as the model-checking engine. In the presence of abstraction and refinement, the tool feeds MCMAS with the abstract model together with the property φ. If model-checker returns true for an ACTLK property, then the tool returns true to the user. Otherwise, the tool automatically checks the generated counterexample based on proposition 4, and reports if it is a real counterexample, which will be returned to the user, or verification needs a refinement round. The tool performs an automated refinement if it is required. For the properties that are discussed in section 8.4, the tool asks user to select a set of invisible local variables to be added to the abstract model for the refinement when model-checker returns true. This will continue until all the related invisible local variables turn to visible, or a valid counterexample is found.

For this section, we choose one temporal and three epistemic properties for the case study of conference paper review system (CRS) with the information leakage vulnerability described in the introduction. We first verify the query (Query 1) "$\mathsf{author}(\mathsf{p}_1, \mathsf{a}_1) \wedge \neg\mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_1) : AG(\neg\mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_1))$" which states that if in the initial states, agent a_1 is the author of paper p_1 and not the reviewer of his own paper, then it is not possible for a_1 to be assigned as the reviewer of his paper p_1. Query 2 "$\neg\mathsf{submittedreview}(\mathsf{p}_1, \mathsf{a}_1) \wedge \mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2) : AG(K_{\mathsf{a}_1}\mathsf{review}(\mathsf{p}_1, \mathsf{a}_2) \to AG(\neg\mathsf{submittedreview}(\mathsf{p}_1, \mathsf{a}_1))$" checks if in the initial states, a_2 is the reviewer of paper p_1 and a_1 has not submitted a review for p_1, then a_1 cannot submit a review for p_1 later if he *reads* the review of a_2 (knowledge by readability). Query 3 "$\mathsf{author}(\mathsf{p}_1, \mathsf{a}_1) : AG(\mathsf{AllPapersAssigned} \wedge \mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2) \to \neg K_{\mathsf{a}_1}\mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2))$" asks if a_1 is the author of p_1, then it is not possible for a_1 to find the reviewer of his paper when his paper is assigned to a_2, which is not ACTLK. Query 4 "$\mathsf{author}(\mathsf{p}_1, \mathsf{a}_1) : AG(\mathsf{AllPapersAssigned} \wedge \mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2) \to K_{\mathsf{a}_1}\mathsf{reviewer}(\mathsf{p}_1, \mathsf{a}_2))$" has ACTLK property, which checks if a_1 can always find who the reviewer of his paper is when all the papers are assigned.

Fig. 4. Comparison of the verification time for the queries 1 and 2 between our tool which uses MCMAS as the model-checking engine, PoliVer and RW



	Concrete model		Abstraction and refinement		
	time(s)	BDD vars	time(s)	Max BDD vars	last ref time
Query 3	6576.5	180	148.3	80	3.28
Query 4	6546.4	180	174.1	98	21

Fig. 5. A comparison of query verification time (in second) and runtime memory usage (in MB) between the concrete model and automated abstraction refinement method

Queries 1 and 2 can be verified in access control policy verification tools like RW and PoliVer, which model knowledge by readability. We compare our tool in the presence of abstraction and refinement with RW and PoliVer from the point of verification time in figure 4. It is important to note that when applying abstraction and refinement, a high percentage of evaluation time is spent on generating abstract models, invoking executable MCMAS which also invokes Cygwin library, and verifying the counterexamples. In most of our experiments, verification of the final abstract model by MCMAS takes less than 10ms.

The novel outcome of our research is the verification of the queries 3 (interactive refinement) and 4 (fully automated refinement) where PoliVer and RW are unable to detect information leakage in CRS policy. In PoliVer and RW, the author never finds a chance to see who the reviewer of his paper is and therefore safety property holds in the system. Modelling in interpreted systems reveals that the author can reason who is the reviewer of his paper. For query 3, the tool also outputs the counterexample which demonstrates the sequence of actions that allows the author to reason about the reviewer of his paper. Figure 5 shows the practical importance of our abstraction method.

10 Conclusion

In this research, we introduced a framework for verifying temporal and epistemic properties over access control policies. In order to verify knowledge by reasoning, we used interpreted systems as the basic framework, and to make the verification practical for medium to large systems, we extended counterexample-guided refinement known as CEGAR to cover safety properties in ACTLK. Case studies and experimental results show a considerable reduction in time and space when abstraction and refinement are in use. We also apply an interactive refinement for some useful properties that does not reside in ACTLK like the ones that contain the negation of knowledge modality.

Acknowledgement. We would like to acknowledge Microsoft Research and EPSRC project TS/I002529/1 “Trust Domains” for funding this research.

References

1. Becker, M.Y.: Specification and analysis of dynamic authorisation policies. In: Proceedings of 22nd IEEE Computer Security Foundations Symposium, CSF (2009)
2. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems through model checking. Journal of Computer Security 16(1), 1–61 (2008)
3. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and Reasoning About Dynamic Access-Control Policies. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 632–646. Springer, Heidelberg (2006)
4. Mardare, R., Priami, C.: Dynamic epistemic spatial logics. Technical report, The Microsoft Research-University of Trento Centre for Computational and Systems Biology (2006)
5. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
6. Fagin, R., Halpern, J.Y., Moses, Y., Vardis, M.Y.: Knowledge-based programs. Distributed Computing 10(4), 199–225 (1997)
7. 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)
8. Clarke, E.M., Lu, Y., Com, B., Veith, H., Jha, S.: Tree-like counterexamples in model checking. In: LICS 2002: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society (2002)
9. Lomuscio, A., Raimondi, F.: MCMAS: A Model Checker for Multi-agent Systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 450–454. Springer, Heidelberg (2006)
10. Koleini, M., Ritter, E., Ryan, M.: Reasoning about knowledge in dynamic policies. Technical report, University of Birmingham, School of Computer Science (2012), http://www.cs.bham.ac.uk/~mdr/research/papers/pdf/13-mc-knowledge.pdf
11. Aucher, G., Boella, G., van der Torre, L.: Privacy Policies with Modal Logic: The Dynamic Turn. In: Governatori, G., Sartor, G. (eds.) DEON 2010. LNCS, vol. 6181, pp. 196–213. Springer, Heidelberg (2010)
12. Koleini, M., Ryan, M.: A Knowledge-Based Verification Method for Dynamic Access Control Policies. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 243–258. Springer, Heidelberg (2011)
13. Cohen, M., Dam, M., Lomuscio, A., Russo, F.: Abstraction in model checking multi-agent systems. In: AAMAS 2009: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, pp. 945–952 (2009)
14. Zhou, C., Sun, B., Liu, Z.: Abstraction for model checking multi-agent systems. Frontiers of Computer Science in China 5, 14–25 (2011)
15. Lomuscio, A., Raimondi, F.: The complexity of model checking concurrent programs against CTLK specifications. In: AAMAS 2006: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 548–550. ACM Press (2006)
16. Cohen, M., Dam, M., Lomuscio, A., Qu, H.: A symmetry reduction technique for model checking temporal-epistemic logic. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence, IJCAI 2009 (2009)
17. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)

Automatic Testing of Real-Time Graphics Systems

Robert Nagy[1], Gerardo Schneider[2], and Aram Timofeitchik[3]

[1] Dfind Redpatch, Sweden
[2] Department of Computer Science and Engineering,
Chalmers | University of Gothenburg, Sweden
[3] DQ Consulting AB, Sweden
ronag89@gmail.com, gerardo.schneider@gu.se, aram.timofeitchik@dqc.se

Abstract. In this paper we deal with the general topic of verification of real-time graphics systems. In particular we present the Runtime Graphics Verification Framework (*RUGVEF*), where we combine techniques from runtime verification and image analysis to automate testing of graphics systems. We provide a proof of concept in the form of a case study, where *RUGVEF* is evaluated in an industrial setting to verify an on-air graphics playout system used by the Swedish Broadcasting Corporation. We report on experimental results from the evaluation, in particular the discovery of five previously unknown defects.

1 Introduction

Traditional testing techniques are insufficient for obtaining satisfactory code coverage levels when it comes to testing real-time graphical systems. The reason for this is that the visual output is difficult to formally define, as it is both dynamic and abstract, making programmatic verification difficult to perform [6]. Inherent properties of real-time graphics, such as non-determinism and time-based execution, make errors hard to detect and reproduce. Furthermore, dependencies such as hardware, operating systems, drivers and other external run-time software also make the task of testing quite difficult, as witnessed by Id Software during the initial release of their video-game Rage, where the game suffered problems with texture artifacts [11]. Even though the software itself performed correctly, the error still occurred when executed on systems with certain graphic cards and drivers.

A common method for verifying real-time graphics is through ocular inspections of the software's visual output. The correctness is manually checked by comparing the subjectively expected output with the output produced by the system. Some disadvantages with this approach are that it requires extensive working hours, it is repetitive, and it makes regression testing practically inapplicable. Moreover, the subjective definition of correctness makes it possible for some artifacts to be recognized as errors by some testers, but not by others [10]. Furthermore, some errors might not be perceptible in the context of specific tests thereby making ocular inspections even more prone to human-error.

In this paper, we present a conceptual model for automatic testing real-time graphics system, with the aim of increasing the probability of finding defects, and making software verification more efficient and reliable. The proposed solution is formalized as the *Runtime Graphics Verification Framework* (*RUGVEF*), based on techniques from runtime verification and image analysis, defining practices and artifacts needed to increase the automation of testing. We implemented and evaluated the framework by using it in the development setting of *CasparCG*, a real-time graphics system used by the *Swedish Broadcasting Corporation* (*SVT*) for producing most of their on-air graphics. We also present an optimized implementation of the image quality assessment technique *SSIM*, which enables real-time analysis of *Full HD* video produced by *CasparCG*. As a result of the application of *RUGVEF* to *CasparCG* we identified 5 previously unknown defects that were not previously detected with existing testing practices at *SVT*, and 6 out of 16 known defects that were injected back into *CasparCG* could be found. This shows that *RUGVEF* can indeed successfully complement existing verification practices by automating the detection of contextual and temporal errors in graphical systems. Using the framework allows for earlier detection of defects and enables more efficient development through automated regression testing. In addition to this, the framework makes it possible to test the software in combination with its external environment, such as hardware and drivers.

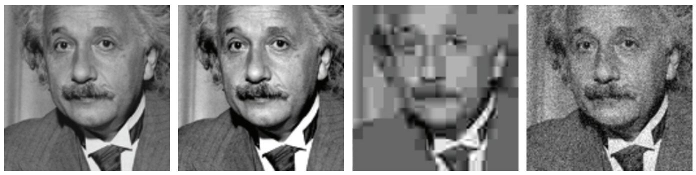
In summary our contributions are: i) The framework *RUGVEF* for automating the testing of real-time graphics systems; ii) The implementation of the framework into a tool, and its application to an industrial case study (*CasparCG*), finding 5 previously unknown defects; iii) An optimized implementation of *SSIM*, an image quality assessment technique not previously applicable to the real-time setting of *CasparCG*.

We start with some background in next section, and we outline our conceptual framework *RUGVEF* in section 3 . We present our case study in section 4, of which we show the results in section 5. Related work is presented in section 6.

2 Background

We give here a very short introduction to runtime verification, and provide a description on some image quality assessment techniques.

Runtime Verification (RV) offers a way for verifying systems as a whole during their execution [2]. The verification is performed at runtime by monitoring system execution paths and states, checking whether any predefined formal logic rules are being violated. Additionally, RV can be used to verify software in combination with user-based interaction, adding more focus toward user specific test-cases, which more likely could uncover end-user experienced defects. However, care should be taken as RV adds an overhead potentially reducing system performance. This overhead could also possibly affect the time sensitivity of systems in such way that they appear to run correctly while the monitor is active, but not after it has been removed, a common problem when checking for e.g. data-races in concurrent execution [2].



(a) MSE:0 SSIM:1.000 (b) MSE:306 SSIM:0.928 (c) MSE:309 SSIM:0.580 (d) MSE:309 SSIM:0.576

Fig. 1. *Image Quality Assessment* of distorted images using *MSE* and *SSIM* [15]

Image Quality Assessment is used to assess the quality of images or video-streams based on models simulating the *Human Visual System* (*HVS*) [15]. The quality is defined as the *fidelity* or similarity between an image and its reference, and is quantitatively given as the differences between them. Models of the *HVS* describe how different type of errors should be weighted based on their *perceptibility*, e.g. errors in *luminance* are more perceptible than errors in *chrominance* [9]. [1] However, there is a trade-off between the accuracy and performance of algorithms that are based on such models.

Binary comparison is a high performance method for calculating image fidelity, but does not take human perception into account. This could potentially cause problems where any binary differences found are identified as errors even though they might not be visible, possibly indicating false negatives.

Another relatively fast method is the *Mean Squared Error (MSE)*, which calculates the cumulative squared difference between images and their references, where higher values indicate more errors and lower fidelity. An alternative version of *MSE* is the *Peak Signal to Noise Ratio (PSNR)* which instead calculates the peak-error (i.e. noise) between images and their references. This metric transforms *MSE* into a logarithmic decibel scale where higher values indicate fewer errors and stronger fidelity. The *MSE* and *PSNR* algorithms are commonly used to quantitatively measure the performance and quality of lossy compression algorithms in the domain of video processing [6], where one of the goals is to keep a constant image quality while minimizing size, a so-called constant rate factor [7]. This constant rate is achieved, during the encoding process, by dynamically assessing image quality while optimizing compression rates accordingly.

Structural Similarity Index (SSIM) is an alternative measure that puts more focus on modeling human perception, but at the cost of heavier computations. The algorithm provides more interpretable relative percentage measures (0.0-1.0), in contrast to *MSE* and *PSNR*, which present fidelity as abstract values that must be interpreted. *SSIM* differs from its predecessors as it calculates distortions in perceived structural variations instead of perceived errors. This difference is illustrated in Fig. 1, where (b) has a uniform contrast distortion over the entire image, resulting in a high perceived error, but low structural

[1] Luminance is a brightness measure and chrominance is about color information.

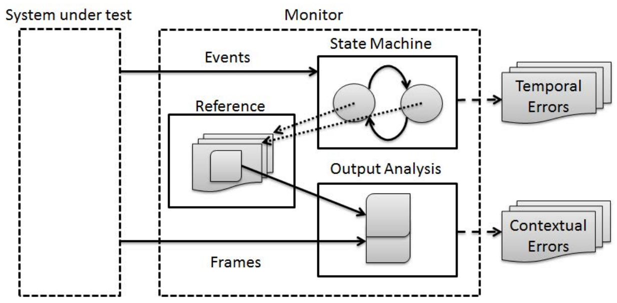


Fig. 2. The *RUGVEF* framework

error. Unlike *SSIM*, *MSE* considers (b), (c), and (d) to have the same image fidelity to the reference (a), but this is clearly not the case due to the relatively large structural distortions in (c) and (d). Tests conducted have shown that *SSIM* provides more consistent results compared to *MSE* and *PSNR* [15]. Furthermore, *SSIM* is also used in some high end applications as an alternative to *PSNR*.[2]

3 The Runtime Graphics Verification Framework

In this section we start by describing the *Runtime Graphics Verification Framework* (*RUGVEF*), for verifying graphics-related system properties. We then state the prerequisites for testing such systems, and finally, we explain how graphical content is analyzed for correctness using image quality assessment.

3.1 The *RUGVEF* Conceptual Model

RUGVEF can be used to enable verification of real-time graphics systems during their execution. Its verification process is composed of two mechanisms: i) checking of execution paths, and ii) verification of graphical output. Together they are used to evaluate *temporal* and *contextual* properties of the system under test. Note that the verification can also include its external runtime environment, such as hardware and drivers.

The verification process, illustrated in Fig. 2, is realized as a monitor application that runs in parallel with the tested system. During this process system and monitor are synchronized through event-based communication where events are used to identify changes in the system's runtime state, thereby verifying the system's temporal correctness. State transitions should always occur when the graphical output changes, allowing legal graphical states to be represented through reference data. These references, either predefined or generated during testing using N-version programming, are in turn used for determining the

[2] http://www.videolan.org/developers/x264.html

correctness of state properties through objective comparisons against graphical output produced by the system using appropriate image assessment techniques.

As an example let us consider testing a simple video player having three system control actions (`play`, `pause`, and `stop`), which according to the specification change from 3 different states: from `Idle` to `Playing` with action `play`. From `Playing` it is possible to go to state `Idle` with action `stop` and to `Paused` with action `pause`; and finally from `Paused` to `Playing` with action `play`, and with `stop` to state `Idle`. In this formal definition (the above gives place to a Finite-State Machine — FSM), transitions are used to describe the consequentiality of valid system occurrences that potentially could affect the graphical output. Thus, as the video player is launched the monitor application is started and initialized to the video player's `Idle` state, specifying during this state that only completely black frames are expected. Any graphical output produced is throughout the verification intercepted and compared against specified references, where any mismatches detected correspond to contextual properties being violated. At some instant, when one of the video player's controls is used, an event is triggered, signaling to the monitor that the video player has transitioned to another state. In this case, there is only one valid option and that is the event signaling the transition from `Idle` to `Playing` state (any other events received would correspond to temporal properties being violated). As valid transitions occur, the monitor is updated by initializing the target state, in this case the `Playing` state, changing references used according to that state's specifications.

3.2 Prerequisites

There is a limitation in using comparisons for evaluating graphical output. To illustrate this consider a moving object being frame-independently rendered at three different rendering speeds, showing that during the same time period, no matter what frame rate is used, the object will always be in the same location at a specific time. The problem is that rendering speeds usually fluctuate, causing consecutive identical runs to produce different frame-by-frame outputs. For instance two runs having the same average frame rate but with varying frame-by-frame results, will make it impossible to predetermine the references that should be used. For this reason, the rendering during testing must always be performed in a time-independent fashion. That is, a moving object should always have moved exactly the same distance between two consecutively rendered frames, no matter how much time has passed.

3.3 Image Quality Assessment for Analyzing Graphical Output

Analysis of graphical output is required in order to determine whether contextual properties of real-time graphics systems have been satisfied. *RUGVEF* achieves this by continuously comparing the graphical output against predefined references. We discuss two separate image quality assessment techniques for measuring the similarity of images: one based on absolute correctness, and the other based on perceptual correctness.

Absolute correctness is assessed using binary comparison, where images are evaluated pixel by pixel in order to check whether they are identical. This technique is effective for finding differences between images that are otherwise difficult or impossible to visually detect, which could for instance occur as a result of using mathematically flawed algorithms. However, it is not always the case that non-perceptible dissimilarities are a problem, requiring in such cases that a small tolerance threshold is introduced in order to ignore acceptable differences. One example of this could occur when the monitored system generates graphical output using a Graphical Processing Unit. based runtime platform, conforming to the *IEEE 754* floating point model[3], while its reference generator is run on a *x86 CPU* platform, using an optimized version of the same model[4], possibly causing minor differences in what otherwise should be binarily identical outputs.

Perceptual correctness is estimated through algorithms based on models of the human visual system, and is used for determining whether images are visually identical. Such correctness makes graphics analysis applicable to the output of physical video interfaces which compresses images into lossy color spaces [15,9], with small effects on perceived quality [9], but with large binary differences.

We have evaluated the three common image assessment techniques, *MSE*, *PSNR*, and *SSIM*, which are based on models of the *HVS*. Although *MSE* and *PSNR* are the most computationally efficient and widely accepted in the field of image processing, we have found *SSIM* to be the best alternative. The reason for this is that *MSE* and *PSNR* are prone to false positives and present fidelity as abstract values that need to be interpreted. As an example, when verifying the output from a physical video interface we found that an unacceptably high error threshold was required in order for a perceptually correct video stream to pass its verification. *SSIM* on the other hand was found to be more accurate, also presenting results as concrete similarity measure given as a percentage (0.0-1.0). Additionally, both *MSE* and *PSNR* have recently received critique due to lacking correspondence with human perception [15]. The main problem with *SSIM* is that current implementations are not efficient.

4 Case Study - *CasparCG*

In order to evaluate the feasibility of our framework, a case study was performed in an industrial setting where we created, integrated, and evaluated a verification solution based on *RUGVEF*. We first describe *CasparCG*, and we then show how testing of *CasparCG* was improved using *RUGVEF*.

4.1 *CasparCG*

The development of *CasparCG* started in 2005 as an in-house project for on-air graphics and was used live for the first time during the 2006 Swedish elections [13]. Developing this in-house system enabled *SVT* to greatly reduce costs

[3] `http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf`

[4] `http://msdn.microsoft.com/en-us/library/e7s85ffb.aspx`

by replacing expensive commercial solutions with a cheaper alternative. During 2008 the software was released under an open-source license, allowing external contributions to the project. *CasparCG* 2.0, was released in April 2012 with the successful deployment in the new studios of the show *Aktuellt* [12]. During broadcasts *CasparCG* renders on-air graphics such as bumpers, graphs, news tickers, and name signs. All graphics are rendered in real-time to different video layers that are composed using alpha blending into a single video-stream. The frame rate is regulated by the encoding system used by the broadcasting facility.

The broad range of features offered by *CasparCG* allows the replacement of several dedicated devices during broadcasting (e.g. video servers, character generators, and encoders), making it a highly critical component as failures could potentially disrupt several stages within broadcasts. The system is expected to handle computationally heavy operations during real-time execution, e.g. high quality deinterlacing[5] and scaling of high definition videos. A single program instance can also be used to feed several video-streams to the same or different broadcasting facilities, requiring good performance and reliability.

CasparCG is incrementally developed and is mainly tested through code reviews and ocular inspections. Code reviews are performed continuously throughout the iterative development, roughly every two weeks and also before any new version release. Reviews usually consist of informal walkthroughs where either the full source code or only recently modified sections are inspected, in order to uncover possible defects. Ocular inspections are performed during the later stages of the iterative development, when *CasparCG* is nearing a planned release. The inspections consist of testers enumerating different combinations of system functionalities and visually inspecting that the output produced looks correct.

Whenever an iteration is nearing feature completion, an alpha build is released, allowing users to test the newly added functionality while verifying that all previously existing features still work as expected. Once an iteration becomes feature complete, a beta build is released that further allows users to test system stability and functionality. As defects are reported and fixed, additional beta builds are released until the iteration is considered stable for its final release. Alpha and beta releases are viewed, by the development team, as a cost-effective way for achieving relatively large code coverage levels, where the assumption is that users try more combinations of features, compared to the in-house testing, and that the most commonly used features are tested the most.

4.2 Verifying *CasparCG* with *RUGVEF*

RUGVEF was integrated into the testing workflow of *CasparCG* with the aim of complementing existing practices (particularly ocular inspections), in order to improve the probability of detecting errors, while maintaining the existing reliability levels of its testing process. In this section, we present our contribution

[5] A process where an *interlaced* frame consisting of two interleaved frames (fields) are split into two full *progressive* frames.

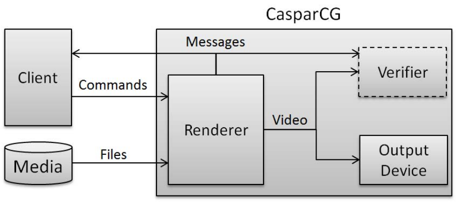


Fig. 3. The verifier is implemented as an output module, running as a part of *CasparCG*

to the testing of *CasparCG*, consisting of two separate verification techniques: local and remote, allowing the system to be verified alternatively on the same and different machine. We also present our optimized *SSIM* implementation, used for real-time image assessment, and a theoretical argumentation on how our approach is indeed an optimization in relation to a reference implementation [4].

Local Verification. During local verification, the verification process is concurrently executed as a plugin module inside *CasparCG*, allowing output to be intercepted without using middleware or code modifications. Fig. 4.2 illustrates that the verifier is running as a regular output module inside *CasparCG*, directly intercepting the graphical output (i.e. video) and the messages produced.

The main difficulty of verifying *CasparCG* is to check its output as it is dynamically composed of multiple layers. Consider the scenario where a video stream, initially composed by one layer of graphics, is verified using references. In this case, the reference used is simply the source of the graphics rendered. However, at some point, as an additional layer is added, the process requires a different reference for checking the stream that now is composed of two graphical sources. The difficulty, in this case, is to statically provide references for each possible scenario where the additional layer has been added on top of the other (as this can happen at any time). As a solution, we instead analyze the graphical output through a reference implementation that mimics basic system functionalities of *CasparCG* (e.g. blending of multiple layers). Using the original source files, the reference implementation generates references at runtime which are expected to be binarily equal to the graphical output of *CasparCG*. The reference implementation only needs to be verified once, unless new functionality is added, as it is not expected to change during *CasparCG*'s development.

Another problem of verifying *CasparCG* lies in defining the logic of the system, where each additional layer or command considered would require an exponential increase in the number of predefined states. For example an FSM representing a system with two layers would only require half the amount of states compared to an FSM representing the same system with three layers. In order to avoid such bloated system definitions, we instead define a generic description of *CasparCG* where one state machine represents all layers which are expected to be functionally equal. This allows temporal properties of each layer to be monitored separately while the reference implementation is used for checking contextual properties of the complete system.

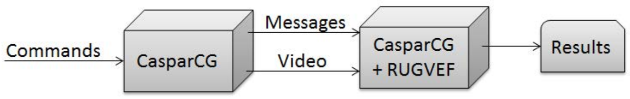


Fig. 4. The remote verification uses two instances of *CasparCG*

The process in local verification is computationally demanding, affecting the system negatively during periods of high load, thus making verification inapplicable during stress-testing. Another limitation identified was that not all components of the system are verifiable; that it is impossible to check the physical output produced by *CasparCG*, which could be negatively affected by external factors (e.g. hardware or drivers). So, in order to more accurately monitor *CasparCG*, with minimal overhead and including its physical output, we further extended our implementation to include remote verification.

Remote Verification. During remote verification, the verifier is executed non-intrusively on a physically different system. Fig. 4 shows this solution, consisting of two *CasparCG* instances running on separate machines, where the first instance receives the commands and produces the output, and the second instance captures the output and forwards it to the *RUGVEF* verification module.

The main problem of remote verification is that the video card interface of *CasparCG* compresses graphical content, converting it from the internal *BGRA* color format to the *YUV420* color format, before transmitting it between the machines. These compressions cause data loss, making binary comparisons inapplicable, instead requiring that the output is analyzed through other image assessment techniques that are based on the human visual system. In this implementation, we chose to use *SSIM*, as it seems to be the best alternative for determining whether two images are perceptually equal. However, the reference *SSIM* implementation [4] is only able to process one frame every few seconds, making real-time analysis of *CasparCG*'s graphical output impossible (as it is produced at a minimum rate of 25 frames per second). In what follows we discuss specific optimizations performed in order to make *SSIM* applicable to the *RUGVEF* verification process of *CasparCG*.

On the Implementation of SSIM. The main challenge of improving the implementation of *SSIM* consisted in achieving the performance that would allow the algorithm to be minimally intrusive while keeping up with data rate of *CasparCG*. The main bottleneck concerning efficiency in current implementations of *SSIM* is the quadratic time complexity, $O(N^2M^2)$, depending on the HDTV resolution (N), and on the window size (M) used in the fidelity measurements. In order to improve the performance, we implemented the algorithm using *Single-Instruction-Multiple-Data* instructions (*SIMD*) [8], allowing us to perform simultaneous operations on vectors of 128 bit values, in this case four 32 bit floating point values using one single instruction. Also, in order to fully utilize *SIMD*, we chose to replace the recommended window size of $M = 11$ in [15] with M=8, allowing calculations to be evenly mapped to vector sizes of four elements (i.e. two vectors per row).

Furthermore, we parallelized our implementation by splitting images into several dynamic partitions, which are executed on a task-based scheduler, enabling load-balanced cache-friendly execution on multicore processors [5]. Dynamic partitions enable the task-scheduler to more efficiently balance the load between available processing units [5], by allowing idle processing units to split and steal sub-partitions from other busy processing units' work queues. Using all processors, we are able to achieve a highly scalable implementation.

The final time complexity achieved by our optimized *SSIM* implementation is $O((N^2\ (M^2+24))/(12p))$, where p is the number of available processing units, allowing *SSIM* calculations to be performed in real-time on consumer level hardware at *HDTV* resolutions.

5 Experimental Results

In this section we show: i) The errors found while verifying *CasparCG* using *RUGVEF*, ii) Previously known defects (injected back into *CasparCG*) we could detect, iii) The improvements in terms of accuracy and performance of our optimized *SSIM* implementation w.r.t the reference implementation.

5.1 Previously Unknown Defects

Using *RUGVEF* we were able to detect five previously unknown defects (presented in the order of their severity, as assessed by the developers), namely: i) Tinted colors, ii) Arithmetic overflows during alpha blending, iii) Invalid command execution, iv) Missing frames during looping, v) Minor pixel errors.

Tinted Colors. Using remote verification, we found a defect where a video transmitted by *CasparCG*'s video interface had slightly tinted colors compared to the original source (i.e. the reference). The error was caused by an incorrect *YUV* to *BGRA* transformation that occurred between *CasparCG* and the video interface. Such problems are normally difficult to detect as both the reference and the actual output looks correct when evaluated separately where differences only are apparent during direct comparisons.

Arithmetic Overflows During Alpha Blending. Using *RUGVEF*, we found that in video streams consisting of multiple layers some small "bad" pixels appear, due to a pixel rounding defect. This defect caused arithmetic overflows during blending operations, producing errors as shown in Fig. 5 (b) (seen as blue pigmentations[6]). Since these errors only occur in certain cases and possibly affecting very few pixels, detection using ocular inspections is a time-consuming process requiring rigorous testing during multiple runs.

[6] In B&W this is seen as the small grey parts in the white central part of the picture.

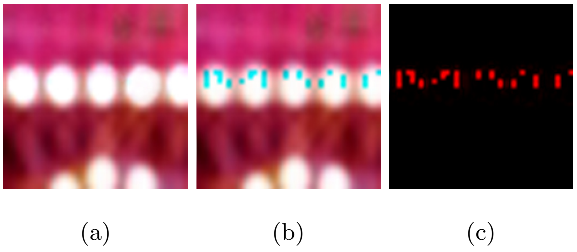


Fig. 5. Pixel rounding defect causing artifact to appear in image (b), highlighted using red color (grey in B&W) in (c), which are not visible in the reference (a)

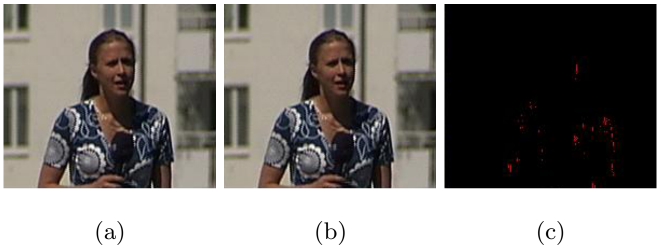


Fig. 6. The output (b) is perceptually identical to the reference (a) while still containing minor pixels errors (c)

Invalid Command Execution. Using *RUGVEF*, we found that the software in certain states accepted invalid commands. For instance it was possible to stop and pause images while in the `Idle` state and to pause while in the `Paused` state. Executing commands on non-existing layers caused unnecessary layers to be initialized, consuming resources in the process. Without *RUGVEF*, this defect would only have been detected after long consecutive system runs, where the total memory consumed would be large enough to be noticed. Furthermore, the execution of these invalid commands produced system responses that indicated successful executions to clients (instead of producing error messages), probably affecting both clients and developers in thinking that this behavior was correct.

Missing Frames During Looping. Using *RUGVEF*, we detected that frames were occasionally skipped when looping videos. The cause of this defect is still unknown and has not been previously detected due to the error being virtually invisible, unless videos are looped numerous times (since only one frame is skipped during each loop).

Minor Pixel Errors. Using local verification, we detected that minor pixel deviations occurred to the output of *CasparCG* that sometimes caused pixel errors of up to 0.8%. These errors are perceptually invisible and could only be detected by using the binary image assessment technique. Fig. 6 shows an example of

Table 1. Previously fixed defects that were injected back into *CasparCG* in order to test whether they are detectable using *RUGVEF*

Rev	Description	Found
N/A	Flickering output due to faulty hardware.	yes
2717	Red and blue color channels swapped during certain runs.	yes
2497	Incorrect buffering of frames for deferred video input.	no
2474	Incorrect calculations in multiple video coordinate transformations.	no
2410	Frames from video files duplicated due to slow file I/O.	yes
2119	Configured RGBA to alpha conversion sometimes not occurring.	yes
1783	Missing alpha channel after deinterlacing.	yes
1773	Incorrect scaling of deinterlaced frames.	no
1702	Video seek not working.	no
1654	Video seek not working in certain video file formats.	no
1551	Incorrect alpha calculations during different blending modes.	no
1342	Flickering video when rendering on multiple channels.	yes
1305	De-interlacing artifacts due to buffer overflows.	no
1252	Incorrect wipe transition between videos.	no
1204	Incorrect interlacing using separate key video.	no
1191	Incorrect mixing to empty video.	no

such a case, where the output in (b) looks identical to the reference in (a) but where small differences have been detected (c).

5.2 Previously Known Defects

In order to evaluate the efficiency of our conceptual model, we injected several known defects into *CasparCG* and tested whether these could be found using *RUGVEF*. The injected defects were mined from the subversion log of *CasparCG* [1] by inspecting the last 12 months of development, scoping the large amount of information while still providing enough relevant defects. In table 1, we present a summary of the gathered defects, where the first column contains the revision id of the log entry, the second a short description of the defect, and the third column indicates whether the defects were possible to detect using *RUGVEF*.

Using *RUGVEF* we were able to detect 6 out of 16 defects that were injected back into *CasparCG*. The defects that could not be found were due to limited reference implementation, which only partially replicated existing functionalities of *CasparCG*. For instance, our reference implementation did not include the scaling of frames or the wipe transition functionalities which made the defects, found in revision 1773 and 1252 respectively, impossible to detect as appropriate references could not be generated.

5.3 Performance of the Optimized SSIM Implementation

We performed our speed improvement benchmarks of our optimized *SSIM* implementation on a laptop computer having 8 logical processing units, each running

Table 2. The optimized *SSIM* implementation compared against a reference implementation at different video resolutions

Implementation	720x576 (SD)	1280x720 (HD)	1920x1080 (Full HD)
Optimized	129 fps	55 fps	25 fps
Reference	1.23 fps	0.55 fps	0.24 fps

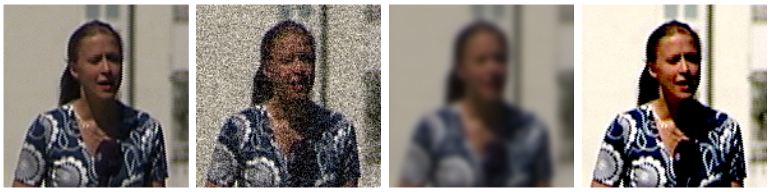


Fig. 7. The results of performing *SSIM* calculation using our optimized implementation (O) and the reference implementation (R) for an undistorted image (a), noisy image (b), blurred image (c), and an image with distorted levels (d)

at 2.0 GHz[7] (which is considerably slower than the target server level computer). Each benchmark consisted of comparing the optimized *SSIM* implementation against the original implementation using the three most common video resolutions, standard definition *(SD)*, high definition *(HD)*, and full high definition *(Full HD)*, by measuring the average time for calculating *SSIM* for 25 randomly generated images. The results of our benchmarks are presented in table 2, showing that our optimized *SSIM* implementation is up to 106 times faster than the original implementation. This increase is larger than the theoretically expected increase of 80 times (calculated using our final time complexity in section 4.2), since our optimized *SSIM* implementation performs all calculations in a single pass, thereby avoiding the memory bottlenecks which existed in the original *SSIM* implementation. Using our implementation, we were able to analyze the graphical output of *CasparCG* in real-time for *Full HD* streams.

Additionally, we also performed an accuracy test by calculating *SSIM* for different distortions in images, comparing the results of our optimized *SSIM* implementation with the results of the original implementation. In Fig. 7, we present the values produced by our optimized *SSIM* implementation "O" and the values produced by the original implementation "R" for the following four types of image distortions: undistorted (a), noisy (b), blurred (c), and distorted levels (d). The result shows that the accuracy of both *SSIM* implementations is nearly identical, as the differences between the values are very small.

[7] Intel Core i7-2630QM.

6 Related Work

The following works address issues related to the testing of graphics: the tool *Sikuli* [16], that uses screenshots as references for automating testing of *Graphical User Interfaces* (*GUI*s); the tool *PETTool* [3], which (semi-) automates the execution of *GUI* based test-cases through identified common patterns; and a conceptual framework for regression testing graphical applications [6]. When it comes to verifying graphical output, the framework in [6] uses a similar approach to *RUGVEF*. However, the tool in [6] focuses on testing system features in isolation, where each test is run separately and targets specific areas of a system (similarly to unit tests). Furthermore, we have also applied our framework to an industrial case study, while there are no indications that something similar has been done in [6], making it difficult to make a detailed comparison.

Finally, the runtime verification tool *LARVA* [2] was used as inspiration source for developing the runtime verification part of *RUGVEF*.

7 Final Discussion

In this paper we have presented *RUGVEF*, a framework for the automatic testing of real-time graphical systems. *RUGVEF* combines runtime verification for checking temporal properties, with image analysis, where reference based image quality assessment techniques are used for checking contextual properties. The assessment techniques presented were based on two separate notions of correctness: absolute and perceptual. We also provided a proof of concept, in the form of a case study, where we implemented and tested *RUGVEF* in the industrial setting of *CasparCG*, an on-air graphics playout system developed and used by *SVT*. The implementation included two separate verification techniques, local and remote, used for verifying the system locally on the same machine with maximal accuracy, and remotely on a different machine, with minimal runtime intrusiveness. Additionally, remote verification allowed the system to be tested as a whole, making it possible to detect errors in the runtime environment (e.g. hardware and drivers). We also created an optimized *SSIM* implementation that was used for determining the perceptual difference between images, enabling real-time analysis of *Full HD* video output produced by *CasparCG*.

When verifying *CasparCG* with *RUGVEF* we identified 5 previously undetected defects. We also investigated whether previously known defects could be detected using our tool, showing that 6 out of 16 injected defects could be found. Lastly, we measured the performance of our optimized *SSIM* implementation, demonstrating a performance gain of up 106 times compared to the original implementation and a negligible loss in accuracy.

Our results show that *RUGVEF* can successfully complement existing verification practices by automating the detection of contextual and temporal errors in graphical systems. Using our framework allows for earlier detection of defects and enables more efficient development through automated regression testing. Unlike traditional testing techniques, *RUGVEF* can also be used to verify the system post deployment. The implementation of the *RUGVEF* tool requires *CasparCG* to run but it should be possible to adapt and apply implementation to other systems as well.[8]

Acknowledgment. We would like to thank Peter Karlsson for his valuable feedback and SVT for giving us the opportunity to develop this work.

References

1. CasparCG (2008), https://casparcg.svn.sourceforge.net/svnroot/casparcg
2. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: SEFM, pp. 33–37. IEEE Comp. Soc. (2009)
3. Cunha, M., Paiva, A.C.R., Ferreira, H.S., Abreu, R.: PETTool: A pattern-based GUI testing tool. In: ICSTE 2010, vol. 1, pp. 202–206 (2010)
4. Distler, T.: Image quality assessment (IQA) library (2011), http://tdistler.com/projects/iqa
5. Farnham, K.: Threading building blocks scheduling and task stealing: Introduction (August 2007), http://software.intel.com/en-us/blogs/2007/08/13/threading-building-blocks-scheduling-and-task-stealing-introduction/
6. Fell, D.: Testing graphical applications. Embedded Sys. Design 14(1), 86 (2001)
7. Li, X., Cui, Y., Xue, Y.: Towards an automatic parameter-tuning framework for cost optimization on video encoding cloud. Int. J. Digit. Multim. Broadc. (2012)
8. Microsoft. Streaming SIMD extensions, SSE (2012), http://msdn.microsoft.com/en-us/library/t467de55.aspx
9. Murching, A.M., Woods, J.W.: Adaptive subsampling of color images. In: ICIP 1994, vol. 3, pp. 963–966 (November 1994)
10. Myers, G.J., Sandler, C.: The Art of Software Testing, 2nd edn. John Wiley & Sons (2004)
11. Sharke, M.: Rage PC launch marred by graphics issues (October 2011), http://pc.gamespy.com/pc/id-tech-5-project/1198334p1.html
12. S.B.C. (SVT). National news: Aktuellt & Rapport, http://www.casparcg.com/case/national-news-aktuellt-rapport
13. S.B.C. (SVT). Swedish election 2006 (2006), http://www.casparcg.com/case/swedish-election-2006
14. Timofeitchik, A., Nagy, R.: Verification of real-time graphics systems. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden (May 2012)
15. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: From error visibility to structural similarity. IEEE Trans. on Image Proc. 13(4), 600–612 (2004)
16. Yeh, T., Chang, T.-H., Miller, R.C.: Sikuli: using GUI screenshots for search and automation. In: UIST 2009, pp. 183–192. ACM (2009)

[8] The project can be downloaded from runtime-graphics-verification.googlecode.com. See [14] for an extended version of the paper.

Equivalence Checking of Quantum Protocols

Ebrahim Ardeshir-Larijani[1,⋆], Simon J. Gay[2], and Rajagopal Nagarajan[3,⋆⋆]

[1] Department of Computer Science, University of Warwick
E.Ardeshir-Larijani@warwick.ac.uk
[2] School of Computing Science, University of Glasgow
Simon.Gay@glasgow.ac.uk
[3] Department of Computer Science, School of Science and Technology, Middlesex University
R.Nagarajan@mdx.ac.uk

Abstract. Quantum Information Processing (QIP) is an emerging area at the intersection of physics and computer science. It aims to establish the principles of communication and computation for systems based on the theory of quantum mechanics. Interesting QIP protocols such as quantum key distribution, teleportation, and blind quantum computation have already been realised in the laboratory and are now in the realm of mainstream industrial applications. The complexity of these protocols, along with possible inaccuracies in implementation, demands systematic and formal analysis. In this paper, we present a new technique and a tool, with a high-level interface, for verification of quantum protocols using equivalence checking. Previous work by Gay, Nagarajan and Papanikolaou used model-checking to verify quantum protocols represented in the stabilizer formalism, a restricted model which can be simulated efficiently on classical computers. Here, we are able to go beyond stabilizer states and verify protocols efficiently on all input states.

Keywords: quantum protocols, equivalence checking, model checking, stabilizers.

1 Introduction

With the emergence of quantum computation and quantum information processing, there is now a need for high level understanding and techniques in the design and analysis of quantum protocols. To this end, we are pursuing a programme of applying formal methods, developed for the analysis of classical computing and communication systems, to analyse and verify quantum systems. The present paper concerns model-checking, in which the behaviour of a system (defined in a formal modelling language) is exhaustively explored in order to verify that a

⋆ Supported by the Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, EPSRC award EP/D063191/1.

⋆⋆ Partially supported by “Process Algebra Approach to Distributed Quantum Computation and Secure Quantum Communication”, Australian Research Council Discovery Project DP110103473.

desired specification is satisfied by all possible execution paths. There are two distinct styles of model-checking. The first style is *property-oriented*, in which a specification is expressed as a logical formula, usually in temporal logic, and the truth of the formula is checked along every possible execution path. The second style is *process-oriented*, in which a specification is expressed as a simple ideal system whose correctness is self-evident, and verification consists of checking that the (model of the) implementation has exactly equivalent behaviour to the specification.

Previous work by Gay, Nagarajan and Papanikolaou [19,24] has developed QMC, a property-oriented model-checking system for quantum protocols. The present paper explores the process-oriented approach. The main novelty is to exploit the fact that quantum operators are *linear*, in the sense of linear algebra, to reduce the number of inputs on which two quantum protocols must be executed in order to check their equivalence. Interpreting quantum protocols as linear operators on a certain vector space, we can check that two protocols denote the same operator by executing them on inputs which form a basis for the space; linearity means that their behaviour on the whole space is determined by their behaviour on a basis. We have implemented a prototype software tool which uses this idea to automatically check the equivalence of two given quantum protocols.

In addition to the usual problem of large state-spaces arising from the possible execution paths and interactions within a system, quantum model-checking presents another challenge. A quantum state on n qubits (quantum bits) is defined by a basis vector expansion involving 2^n complex coefficients, so representing a quantum state as a classical data structure appears to require exponential space. Indeed, much of the interest in quantum computing arises from the fact that in general, quantum systems cannot be efficiently simulated by classical computers. To avoid this problem, we work with the *stabilizer formalism* [1], which allows efficient classical simulation of a restricted set of quantum states and operations on them. Although not sufficient for general-purpose quantum computing, stabilizer states support many interesting quantum protocols such as teleportation [7], superdense coding [8], and key distribution [6], as well as the essential quantum phenomenon of entanglement. The QMC system [19,24] also uses the stabilizer formalism.

We can explain the advantages of the tool described in the present paper, in comparison with QMC, by considering the problem of verifying a quantum teleportation protocol. Quantum teleportation transfers an unknown quantum state from one physical carrier to another, by carrying out a certain sequence of operations. Its specification is that it should be equivalent to the identity operator on a single qubit. To verify teleportation with QMC, first the condition that the output state is the same as the input state is expressed in a property-oriented style. Then the protocol is executed with every one-qubit stabilizer state (there are six of them) as input. Correctness on all of these inputs is interpreted as evidence for, although not absolute proof of, correctness of the protocol on arbitrary inputs. The equivalence checker described in the present paper executes the teleportation protocol on a set of stabilizer states that form a basis for the appropriate vector space; this involves only four states, and correspondingly less computation. Moreover, by linearity, correctness on these four states guarantees that the protocol is correct for arbitrary inputs. Because QMC tests the protocol on these four states (as well as others), we can retrospectively see that QMC also guaranteed correctness, assuming that the protocol satisfies the semantic conditions that we introduce in Section 4.

The remainder of the paper is organised as follows. In Section 2, we give all the necessary preliminaries for our equivalence checking method. In Section 3, we introduce the language QPL with its syntax and a summary of its semantics. We also present some examples of quantum protocols written in QPL. In Section 4, we explain how our equivalence checker works and give details of the implementation. In Section 5, we present some results comparing our equivalence checker with the QMC system, in terms of running time. Finally, in Sections 6 and 7, related work and future research directions are discussed.

2 Technical Foundations

The unit of quantum information is a *qubit* (quantum bit). A vector space equipped with an *inner product* is called Hilbert space.[1] The *state* of a qubit is a vector in the Hilbert space and is specified by $|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ are amplitudes and $|\alpha|^2 + |\beta|^2 = 1$. We use Dirac's notation to denote unit vectors $|0\rangle$ and $|1\rangle$. States are transformed by unitary linear operators in Hilbert space. An interesting quantum operation is *measurement*, which is not unitary. The outcome of measuring the above state $|\Psi\rangle$ is the classical bit 0 with probability $|\alpha|^2$ or 1 with probability $|\beta|^2$. Moreover measurement changes the state of the qubit permanently to $|0\rangle$ or $|1\rangle$. The quantum circuit model is similar to the classical circuit model, except that there are quantum gates acting on qubits. Quantum circuits are usually described in the following way: each line (wire) represents a qubit and boxes represent quantum gates and also measurement. There are also two-qubit gates, which act on two qubits at the same time, for example, *controlled* gates. Each controlled gate consist of a control qubit (depicted by a point) and target qubit (depicted by a circle). If the value of the control qubit is one, then the corresponding unitary gate is applied. In the quantum circuit, control qubit and target qubit are connected by a vertical line. After measurement, the outcome is classical and this is denoted by a double line. For example, *quantum teleportation* [7] is a protocol which transmits a quantum state from a sender to a receiver using a classical channel and an *entangled* pair (i.e. the qubit is not physically transmitted but it is teleported). The circuit which implements teleportation is illustrated in Figure 1. This circuit uses X (not), Z (phase shift), H (Hadamard) and controlled-not (controlled-X) gates:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \ Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \ H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

[1] There are other conditions, which will not concern us.

Prior to the execution of teleportation protocol, two parties (we call them Alice and Bob) share an entangled pair which can be prepared by applying a Hadamard and a controlled-not gate.

The protocol proceeds as follows: Alice combines the qubit to be teleported with her part of the entangled pair, by applying a controlled-not gate followed by a Hadamard gate. Then she measures the qubits in her possession. If the outcome is one, then she applies Z or X to the corresponding qubit (see double lines in Figure 1). Now Bob's part of the entangled pair ends up in the same state as Alice, which demonstrates that a qubit has been successfully transferred from Alice to Bob.

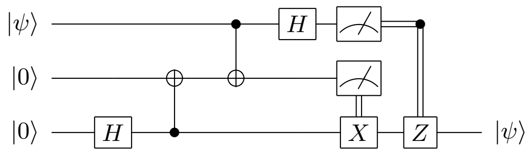


Fig. 1. Teleportation

Stabilizer states are a small but useful subset of quantum states which can be represented in polynomial space [1]. The main idea of the stabilizer formalism is to represent a quantum state $|\phi\rangle$, not by 2^n complex amplitudes (here n is the dimension of $|\phi\rangle$) but by a *stabilizer group*, $Stab(|\phi\rangle)$. This group can be represented by its set of generators G_i, $i \leq n$ such that $G_i |\phi\rangle = |\phi\rangle$. For example the two qubit entangled state $|\phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is represented by $\{X \otimes X, Z \otimes Z\}$:

$$X \otimes X |\phi\rangle = |\phi\rangle$$
$$Z \otimes Z |\phi\rangle = |\phi\rangle$$

More importantly the effects of a certain class of operations and *measurement* on the stabilizer states can be described by a polynomial time algorithm:

Theorem 1. *(Gottesman-Knill, [23, p. 464]) Any quantum computation which consists of only the following components:*

1. *State preparation, Hadamard gates, Phase gates, Controlled-Not gates and Pauli gates.*
2. *Measurement gates.*
3. *Classical control conditions on the outcomes of measurements.*

can be efficiently simulated on a classical computer.

The notion of *density operator* was introduced by Von Neumann and in fact the whole quantum mechanics can be rewritten in the language of density operators.

Let $\{(|\phi_i\rangle, p_i)\}$ denote an ensemble of quantum states (the system is in the state $|\phi_i\rangle$ with probability $|p_i\rangle$). The density operator ρ can be defined by

$$\rho := \sum_i p_i |\phi_i\rangle \langle\phi_i|$$

Here $|\phi_i\rangle \langle\phi_i|$ denotes the outer product of $|\phi_i\rangle$. When $p_i = 1$ we say the state is pure; otherwise the state is mixed. It is useful to note that the density operator ρ is a Hermitian operator: $\rho^\dagger = \rho$ (here $\dagger$ denotes transpose of the complex conjugate) and has two properties. It is positive (for any state $|\varphi\rangle$: $\langle\varphi| \rho |\varphi\rangle \geq 0$) and has a trace condition $Tr(\rho) = 1$. Linear transformations of the form $F : \rho \to \rho'$ where ρ and ρ' are density operators, are called *superoperators*. Suppose we have two systems A and B. Let $|a_1\rangle$, $|a_2\rangle$, $|b_1\rangle$ and $|b_2\rangle$ be any vectors in the state space of A and B. The *partial trace* [23, page 105] of the composite system AB is defined by:

$$Tr_B(|a_1\rangle \langle a_2| \otimes |b_1\rangle \langle b_2|) \equiv |a_1\rangle \langle a_2|\, Tr(|b_1\rangle \langle b_2|)$$

The mathematical interpretation of a quantum information processing system, which is given some quantum input and produces some quantum output, is a linear operator. This is very specific to quantum systems and has no analogue in classical computing. In particular, quantum systems with no measurement can be abstracted by unitary operators, which are linear. In order to check a property of such system, it is sufficient to examine the standard basis of Hilbert space (e.g. for a system operating on one qubit, we check only $|0\rangle$ and $|1\rangle$ and because of linearity we can extend our argument to any state of the general form $\alpha|0\rangle + \beta|1\rangle$). In the case where quantum systems involve measurement, the mathematical interpretation is superoperators, instead of unitaries. Superoperators operate on the space of density matrices, with dimension 2^{2n} for n qubits. Then the behaviour of a quantum system with measurement can be examined by a basis of the space of density matrices. However, in general, for verification of such quantum systems (especially using model-checking), it is impossible to specify and manipulate quantum states on classical computers because there is a continuum of quantum states. Therefore, we use stabilizer states which we can manipulate efficiently. The following theorem [17] finds a stabilizer basis for the space of density matrices, which we shall use later for equivalence checking.

Theorem 2. *The space of density matrices for n-qubit states, considered as a $(2^n)^2$-dimensional real vector space, has a basis consisting of density matrices of n-qubit stabilizer states.*

Notation 1. *Write the standard basis for n-qubit states as $\{|x\rangle \mid 0 \leqslant x < 2^n\}$, considering numbers to stand for their n-bit binary representations. We omit normalization factors when writing quantum states. With this notation, for $n \geqslant 1$ let $\mathsf{GHZ}_n = |0\rangle + |2^n - 1\rangle$ and $\mathsf{iGHZ}_n = |0\rangle + i\,|2^n - 1\rangle$, as n-qubit states.*

Lemma 1. *For all $n \geqslant 1$, GHZ_n and iGHZ_n are stabilizer states.*

Proof. By induction on n. For the base case ($n = 1$), we have that $|0\rangle + |1\rangle$ and $|0\rangle + i\,|1\rangle$ are stabilizer states, by applying H and then P to $|0\rangle$.

For the inductive case, GHZ_n and iGHZ_n are obtained from $\mathsf{GHZ}_{n-1} \otimes |0\rangle$ and $\mathsf{iGHZ}_{n-1} \otimes |0\rangle$, respectively, by applying CNot to the two rightmost qubits. □

Lemma 2. *If $n \geqslant 1$ and $0 \leqslant x, y < 2^n$ with $x \neq y$ then $|x\rangle + |y\rangle$ and $|x\rangle + i\,|y\rangle$ are stabilizer states.*

Proof. By induction on n. For the base case ($n = 1$), the closure properties imply that $|0\rangle + |1\rangle$, $|0\rangle + i\,|1\rangle$ and $|1\rangle + i\,|0\rangle$ (equivalent to $|0\rangle - i\,|1\rangle$ by scalar multiplication) are stabilizer states.

For the inductive case, consider the binary representations of x and y. If there is a bit position in which x and y have the same value b, then $|x\rangle + |y\rangle$ is the tensor product of $|b\rangle$ with an $(n-1)$-qubit state of the form $|x'\rangle + |y'\rangle$, where $x' \neq y'$. By the induction hypothesis, $|x'\rangle + |y'\rangle$ is a stabilizer state, and the conclusion follows from the closure properties. Similarly for $|x\rangle + i\,|y\rangle$.

Otherwise, the binary representations of x and y are complementary bit patterns. In this case, $|x\rangle + |y\rangle$ can be obtained from GHZ_n by applying X to certain qubits. The conclusion follows from Lemma 1 and the closure properties. The same argument applies to $|x\rangle + i\,|y\rangle$, using iGHZ_n. □

Proof of Theorem 2. This is the space of Hermitian matrices and its obvious basis is the union of

$$\{|x\rangle\langle x| \mid 0 \leqslant x < 2^n\} \tag{1}$$

$$\{|x\rangle\langle y| + |y\rangle\langle x| \mid 0 \leqslant x < y < 2^n\} \tag{2}$$

$$\{-i\,|x\rangle\langle y| + i\,|y\rangle\langle x| \mid 0 \leqslant x < y < 2^n\}. \tag{3}$$

Now consider the union of

$$\{|x\rangle\langle x| \mid 0 \leqslant x < 2^n\} \tag{4}$$

$$\{(|x\rangle + |y\rangle)(\langle x| + \langle y|) \mid 0 \leqslant x < y < 2^n\} \tag{5}$$

$$\{(|x\rangle + i\,|y\rangle)(\langle x| - i\,\langle y|) \mid 0 \leqslant x < y < 2^n\}. \tag{6}$$

This is also a set of $(2^n)^2$ states, and it spans the space because we can obtain states of forms (2) and (3) by subtracting states of form (4) from those of forms (5) and (6). Therefore it is a basis, and by Lemma 2 it consists of stabilizer states. □

***Equality test*:** States in the stabilizer formalism are represented by sets of Pauli generators. This representation is not unique since different sets of generators can produce the same state. Therefore a direct comparison of generators cannot establish the equality of two stabilizer states. To check equality of two stabilizer states, which we will require later, we check the linear independence of their corresponding set of generators. If two sets of generators are independent, then indeed they are not equal; otherwise they are equal. Let $|\phi\rangle$ and $|\psi\rangle$ be stabilizer states and $Stab(|\phi\rangle)$, $Stab(|\psi\rangle)$ be their stabilizer groups. It is easy to show that:

Lemma 3

$$|\phi\rangle = |\psi\rangle \iff Stab(|\phi\rangle) = Stab(|\psi\rangle)$$

Proposition 1. *There is a polynomial time algorithm which decides for any stabilizer states $|\phi\rangle$ and $|\psi\rangle$, whether or not $|\phi\rangle = |\psi\rangle$.*

Proof. From Lemma 3 we have $Stab(|\phi\rangle) = Stab(|\psi\rangle) \implies |\phi\rangle = |\psi\rangle$. So it suffices to show $Stab(|\phi\rangle) \subseteq Stab(|\psi\rangle)$ and $Stab(|\phi\rangle) \supseteq Stab(|\psi\rangle)$. If generators of the group $Stab(|\phi\rangle)$ are linearly dependent on the generators of $Stab(|\psi\rangle$ then $Stab(|\phi\rangle) \subseteq Stab(|\psi\rangle)$. To check this, first we represent each $Stab(|\psi\rangle)$ then $Stab(|\phi\rangle)$ by their stabilizer array, an $m \times n$ matrix of Pauli operators, where n is the number of qubits and m is the number of generators of the stabilizer group. Now we consider elementary row operations on the stabilizer array [3]. Here we have two operations: row transpose and row multiplication. These operations do not alter the stabilizer group and hence do not change stabilizer states. In the case of row multiplication, the generators of the stabilizer are altered. Using these two operations a normal form, *Row Reduced Echelon Form (RREF)*, is introduced [3]. It is also shown in [3] that dependencies of stabilizer generators result in I rows in RREF form. We use this result in the following way: we form a combined stabilizer array consisting of generator sets of $Stab(|\psi\rangle$ then $Stab(|\phi\rangle)$ and apply the RREF algorithm on the combined array. The dependencies between generators result in I rows in the combined array. If the number of I rows in the RREF form of combined array is equal to the size of each generator set, then these two sets are dependent. Otherwise they are independent and hence produce different states. The complexity of the RREF algorithm is $O(n^3)$ [3]. □

3 The Language

Many languages have already been proposed for quantum programming; for a survey see [16]. Depending on the underlying model of quantum computation, these languages are designed in different ways. In this paper, we use Selinger's *Quantum Programming Language(QPL)* [25]. This language assumes *QRAM* [20] as a realistic model of quantum computation and follows the slogan of "classical control over quantum computation". Also, QPL has a functional programming style.

In the following we give the textual and structured syntax of QPL (Figure 2). Here, we have a new type **qbit** which stands for qubits variables (for complete typing rules see [25]). Furthermore, we have *unitary operators* on qubits and *measurements*. In the case of qubits, *discard x* means deallocation of qubits which we interpret as *partial trace* of qubits in a composite system. QPL has many useful high-level features like recursive procedures, structured data types and loops. We can formalise different quantum protocols as well as quantum

```
P,Q ::= newbit b|newqbit q:=0 | discard x
skip|P;Q|q*= S|
if b then P else Q end| measure q then P else Q end |
while b do P | proc X:{P} in Q | call X
```

Fig. 2. QPL Syntax

algorithms in QPL. For our equivalence checking, we formalise a protocol at different levels of abstraction and then we check they are equivalent. Typically, for each protocol we specify two quantum programs; one corresponding to its specification and the other to its implementation.

Example 1. **Teleportation**. We have discussed this protocol in Section 2, in the circuit model, and it is depicted in Figure 3. At the specification level, we can think of teleportation as a protocol which transfers the state of a qubit from Alice to Bob. At the implementation level, we apply different operations of the protocol on Alice's qubit and quantum resources (entangled pair) and Bob is able to recover the state of the qubit. The specification and implementation are shown in Figure 3.

Remark 1. This model of teleportation (circuit model and sequential QPL) does not show the physical separation of Alice and Bob. Extending our approach to a concurrent language with communication is a topic for future work.

```
program Teleportation_Specification
input  q0:qbit
output q0:qbit
```

```
program Teleportation_Implementation
input q0:qbit
//Preparing EPR pair.
newqbit q1;
newqbit q2;
q1*=H;
q1q2*=CNot;
//Entangling Alice's qubit.
q0q1*=CNot;
q0*=H;
//Alice's Measurement and Bob's corrections.
measure q0 then q2*=Z else q2*=I end;
measure q1 then q2*=X else q2*=I end
output q2:qbit
```

Fig. 3. Teleportation: Specification and Implementation

Example 2. **Bit Flip Error Correction Code** [23, p. 427]. In this protocol Alice sends her qubit to Bob over a noisy channel and the effect of noise is flipping qubits (by applying the Pauli gate X to random qubits). The implementation of this protocol has three phases: encoding the qubit, sending it over a noisy channel (applying random X) and recovery. The specification and implementation are shown in Figure 4).

```
program Error_Correction_Specification
input  q0:qbit
output q0:qbit
```

```
program Error_Correction_Implementation
input q0:qbit
//Encoding
newqbit q1; newqbit q2;
q0q1*=CNot; q0q2*=CNot;
//Random noise: either do nothing, or apply X to one of q0,q1,q2
newqbit q3; newqbit q4;
q3*=H; q4*=H;
measure q3 then {measure q4 then {q0*=X} else { } end} else end;
measure q3 then else {measure q4 then q1*=X else q2*=X end} end;
//Bob detects the error syndrome and corrects errors
newqbit q5; newqbit q6;
q0q5*=CNot; q1q5*=CNot;
q0q6*=CNot; q2q6*=CNot;
measure q5 then {measure q6 then q0*=X else q1*=X end} else end;
measure q5 then else {measure q6 then q2*=X else q0*=I end} end;
//Bob recovers Alice's qubit
q0q1*=CNot; q0q2*=CNot;
output q0:qbit
```

Fig. 4. Error Correction: Specification and Implementation

The significance of QPL lies in its semantics [25]. It admits a denotational semantics in terms of superoperators. This means that the input and output of quantum programs can be in mixed states and the effect of executing a quantum program can be elegantly described by a superoperator, operating on the density matrices of the input.

Let $D_n = \{A \in \mathbb{C}^{n\times n} | \text{A is positive hermitian and Tr(A)=1}\}$. The Löwner partial order for D_n is defined in the following way: if $A \sqsubseteq B$ then $B - A$ is positive. The domain of denotations for QPL, $(D_n, \sqsubseteq)$, is a poset and a complete partial order (cpo) [25]. Now, the formal semantics of a program in QPL can be defined by a superoperator F of the form: $F : (D_n, \sqsubseteq) \to (D_n, \sqsubseteq)$. For more details about the formal semantics, as well as a static type system for QPL and several examples, see [25].

Remark 2. In the usual definition of density matrices we have that the trace is equal to 1. But in [25] this has been relaxed to $\leqslant 1$ in order to handle infinite loops. However, in this version of equivalence checker we only deal with protocols without loops, so the original definition is sufficient here.

4 The Equivalence Checker

Given QPL programs P_1 and P_2, representing the specification and implementation of a protocol, we want to check their equivalence: $P_1 \cong P_2$. By definition, this means $S_1 = S_2$, where S_i is the superoperator denoted by P_i.

$S_1 = S_2$ means $\forall \rho . S_1(\rho) = S_2(\rho)$, where ρ ranges over all (mixed) quantum states in the input space. By linearity, this is equivalent to $\forall \rho \in \mathfrak{B} . S_1(\rho) = S_2(\rho)$, where $\mathfrak{B}$ is a basis for the input space (and we choose a basis consisting of stabilizer states).

Because a QPL program may contain measurement operators, and quantum measurements have probabilistic results in general, executing P_i on an input ρ leads to a number of possible paths, ranged over by j, and the output is a weighted sum of the final state of execution along each path:

$$S_i(\rho) = \sum_{ij} p_{ij} \left|\varphi_{ij}\right\rangle \left\langle\varphi_{ij}\right|$$

where the p_{ij} are probabilities.

To avoid explicitly representing and computing these weighted sums, we require (and check) that P_1 and P_2 define deterministic functions. This means that for each input ρ we compute the output state $S_i(\rho)^{(j)}$ for each branch j, and check that they are all equal. If they are all equal then we write $S_i(\rho)$ for the common value.

What our equivalence checker outputs, given P_1 and P_2, is the value of the following (informal) expression:

$$\begin{aligned} &\forall \rho \in \mathfrak{B}.\ \forall j,k.\ S_1(\rho)^{(j)} = S_1(\rho)^{(k)} \\ \wedge\ &\forall \rho \in \mathfrak{B}.\ \forall j,k.\ S_2(\rho)^{(j)} = S_2(\rho)^{(k)} \\ \wedge\ &\forall \rho \in \mathfrak{B}.\ S_1(\rho) = S_2(\rho) \end{aligned}$$

Let $\mathit{paths}(P, s)$ denote the set of possible paths, indexed by integers from 1 upwards, when executing program P on input state s. Let $\mathit{StabSim}(P, s, j)$ denote the final state produced by the stabilizer simulation algorithm as in [1], starting with input state s and executing path j of program P. Let $EQ_S(v, w)$ be the equality test algorithm from Section 2. Then the above procedure corresponds to the algorithm in Figure 5.

Remark 3. The overall complexity of the above algorithm is $O(2^{2n} \mathit{poly}(m+n))$, where n is the number of input qubits and m is the number of qubits inside the programs (i.e those created by **newqbit**).

We have implemented our equivalence checker in Java. The compiler for the specification language (QPL) is produced using SableCC [15]. We used a Java implementation of Aaronson-Gottesman's algorithm for interpreting QPL in the stabilizer formalism [24]. The main components of our tool are the following:

– QPL parser (by specifying QPL grammer for SableCC).
– QPL Interpreter/Simulator (using stabilizer array algorithms and their implementation [1,24]).
– Basis Generator (Based on Theorem 2, generates all basis states constructively)
– Equality Test for States (Based on Proposition 1).
– Quantum Measurement Scheduler (to instruct the interpreter to explore all execution paths, arising from quantum measurements).

Remark 4. The result of our equivalence checker is whether a protocol satisfies its specification on all inputs. Therefore, it stands as a proof of correctness of the protocol.

```
for all $v \in \mathfrak{B}$ do
   for all $i \in \{1, 2\}$ do
      $|\phi_i^v\rangle = StabSim(P_i, v, 1)$
      for all $j \in paths(P_i, v) - \{1\}$ do
         if $\neg EQ_S(StabSim(P_i, v, j), |\phi_i^v\rangle)$ then
            return $P_i$ non-deterministic
         end if
      end for
   end for
   if $\neg EQ_S(|\phi_1^v\rangle, |\phi_2^v\rangle)$ then
      return $P_1 \ncong P_2$
   end if
end for
return $P_1 \cong P_2$
```

Fig. 5. Algorithm for checking equivalence of QPL programs

5 Results

We now present some initial experimental results, comparing our equivalence checking technique with the QMC model-checking system. We performed the experiments on a 2.1 GHz Intel Dual Core machine with 3.7 GB RAM, running Windows.

The main results use the examples from Section 3: teleportation (Figure 3) and error-correction (Figure 4). The timings are shown in Figure 6. In both cases our equivalence checker is faster, although the improvement is much smaller for the error-correction example than for the teleportation example.

Protocol	equivalence checking (this paper)	QMC [19,24]
Teleportation	21	75
Quantum error correction	63	72

Fig. 6. Running times in milliseconds for equivalence checking and model-checking (QMC) quantum protocols

```
program Simple-Coin-Toss_Specification
input  q0:qbit
output q0:qbit
```

```
program Simple-Coin-Toss_Implementation
input q0:qbit
//Applying H to q0 creates a superposition in some cases
q0*=H;
measure q0 then q0*=I else q0*=I end
output q0:qbit
```

Fig. 7. Simple-Coin-toss: Implementation

To observe the effect of non-determinism in quantum systems, consider the simple QPL program in Figure 7, which simulates a coin-toss by using the random results of measuring a quantum superposition. If the input state is such that applying H produces a superposition state, then the measurement has two possible outcomes which occur with equal probability. The output of this program is therefore not a deterministic function of its input, and so it is rejected by our equivalence checker, independently of the specification program. Detecting non-determinism of this example takes 14ms.

6 Related Work

The most closely related work is the QMC system [19,24], with which we compared our equivalence checker in Section 5. Both QMC and our equivalence checker are based on the stablizer formalism. There are two main differences. First, QMC uses a more general modelling language which supports concurrency and synchronous communication on channels. Extending our system to a concurrent language is a topic for future work. Second, QMC is property-oriented, using *Exogenous Quantum Propositional Logic (EQPL)* [22] and its temporal extension *Quantum Computation Tree Logic (QCTL)* [4] to express specifications. However, the existing applications of QMC have not used the full power of these logics.

Recently, Feng *et al.* [13] have studied model checking of quantum systems using quantum Markov chains. In their setting, a transition system is determined by set of states consisting of density matrices and transitions in terms of superoperators. They considered model checking of an extension of CTL to the quantum case, using quantum Markov chains. In particular, in the presence of maximal entanglement, they need to compute *accumulated superoperators* [13] from Markov chains. Their paper establishes the foundations for an approach to quantum model checking but we are not aware that they have implemented it.

An alternative style of modelling language is given by quantum process calculus, which has been developed by Gay and Nagarajan [18] (CQP) and Ying *et al.* [27] (qCCS). Bisimulation-based equivalences have been studied by Davidson [10] for CQP and by Feng *et al.* [14] for qCCS. These equivalences provide a foundation for process-oriented specification and verification of quantum systems, but they have not yet been developed into tools.

For synthesis of quantum circuits, Hayes *et al.* [26] introduced *Quantum Information Decision Diagrams (QUIDD)*, which extend *Binary Decision Diagrams(BDD)* [9] to the representation and evaluation of quantum circuits. This technique has been implemented in a tool called QuIDD Pro [26] and applied to many examples. The input of QuIDD Pro is a quantum circuit which is then represented by a QUIDD. This is in contrast with QMC and our approach, which use higher level modelling languages amenable to programming.

Abramsky and Coecke [2] started an extensive line of research on a graphical calculus for reasoning about quantum protocols. Diagrammatic reasoning is supported by an underlying categorical semantics. By using graph rewriting techniques, this idea has been implemented in the tool Quantomatic [12]. We have not compared execution times between Quantomatic and our system, because the input formats are so different (textual vs. graphical). A detailed comparison of the Quantomatic approach and our approach would be an interesting topic for future work.

Belardinelli *et al.* [5] introduced a technique for the verification of quantum protocols using a classical model checker for multi-agent systems, MCMAS [21]. They used the framework of D'Hondt and Panangaden [11] to specify protocols with respect to epistemic properties, implemented a compiler to translate the epistemic description of protocols to the input language of MCMAS. However, their technique represents quantum states by their matrix representation, which imposes scalability restrictions, and it also does not support classical control flow in the protocols.

7 Conclusion and Future Work

We have demonstrated a new approach to the verification of quantum protocols by equivalence checking. We used the stabilizer formalism and its efficient algorithms to represent and manipulate quantum states. This enabled us to develop an equivalence checker for quantum protocols. Using Theorem 2, we were able to take the further step to prove the correctness of protocols for all inputs, not just inputs that are stabilizer states. This provides stronger results than the original conclusions from the stabilizer-based model-checking system QMC, when specifications are expressed in terms of input/output behaviour. Our implementation is also faster than QMC on the examples that we have tested.

There is an important point to make in comparison with QMC. QMC does not require a program to denote a determinstic function, so it is more general, and in cases in which the program is not a deterministic function, the fact that QMC checks it on all stabilizer states as inputs can be interpreted as evidence for correctness. By Theorem 2, in cases when the program is a deterministic function, our equivalence checker gives a guarantee of correctness for all inputs. This result can be retrospectively applied to some QMC verifications, including the examples from Section 5, and could be used to speed up QMC.

The main area for future work is the extension of our techniques to concurrent systems. The idea is to allow a system to be constructed from communicating concurrent components, but still require its overall input/output behaviour to be a deterministic function. This requires extension of the syntax of QPL to a concurrent language, and an argument that every possible interleaving gives a sequentialized system which still has a superoperator semantics and can therefore be analyzed by the same techniques that we have used in this paper. Extending our language and system in this way will support more realistic models of quantum protocols, in which the participants are represented by separate concurrent processes and communication is explicit.

Because we are working within the stabilizer formalism, we can only analyze protocols whose operations are restricted to those allowed in the stabilizer formalism (Theorem 1). There are techniques for extending the stabilizer formalism to a limited number of more general operations and states (for example, [1]) and we would like to investigate those techniques in the context of our equivalence checker. Finally, there is scope for extending the classical aspects of our modelling language and for improving the efficiency of the tool.

Acknowledgement. We would like to thank Nick Papanikolaou for useful discussions and for making the source code of QMC available.

References

1. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. Phys. Rev. A 70, 052328 (2004)
2. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, pp. 415–425 (2004)
3. Audenaert, K.M.R., Plenio, M.B.: Entanglement on mixed stabilizer states: normal forms and reduction procedures. New Journal of Physics 7(1), 170 (2005)
4. Baltazar, P., Chadha, R., Mateus, P.: Quantum computation tree logic—model checking and complete calculus. International Journal of Quantum Information 6(2), 219–236 (2008)
5. Belardinelli, F., Gonzalez, P., Lomuscio, A.: Automated verification of quantum protocols using MCMAS. In: Proc. QAPL. EPTCS, vol. 85, pp. 48–62 (2012)
6. Bennett, C.H., Brassard, G.: Quantum cryptography: Public key distribution and coin tossing. In: Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing, pp. 175–179 (1984)

7. Bennett, C.H., Brassard, G., Crépeau, C., Jozsa, R., Peres, A., Wootters, W.K.: Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. Phys. Rev. Lett. 70, 1895–1899 (1993)
8. Bennett, C.H., Wiesner, S.J.: Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. Phys. Rev. Lett. 69, 2881–2884 (1992)
9. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
10. Davidson, T.A.S.: Formal Verification Techniques Using Quantum Process Calculus. PhD thesis, University of Warwick (2011)
11. D'Hondt, E., Panangaden, P.: Reasoning About Quantum Knowledge. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 553–564. Springer, Heidelberg (2005)
12. Dixon, L., Duncan, R.: Graphical reasoning in compact closed categories for quantum computation. Annals of Mathematics and Artificial Intelligence 56(1), 23–42 (2009)
13. Feng, Y., Yu, N., Ying, M.: Model checking quantum Markov chains. arXiv:1205.2187 (2012)
14. Feng, Y., Duan, R., Ying, M.: Bisimulation for quantum processes. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 523–534. ACM (2011)
15. Gagnon, E.: SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University (1998)
16. Gay, S.J.: Quantum programming languages: survey and bibliography. Mathematical Structures in Computer Science 16(4), 581–600 (2006)
17. Gay, S.J.: Stabilizer states as a basis for density matrices. arXiv:1112.2156 (2011)
18. Gay, S.J., Nagarajan, R.: Communicating Quantum Processes. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 145–157. ACM (2005)
19. Gay, S.J., Nagarajan, R., Papanikolaou, N.: QMC: A Model Checker for Quantum Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 543–547. Springer, Heidelberg (2008)
20. Knill, E.: Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory (1996)
21. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009)
22. Mateus, P., Sernadas, A.: Weakly complete axiomatization of exogenous quantum propositional logic. Information and Computation 204(5), 771–794 (2006)
23. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press (2000)
24. Papanikolaou, N.: Model Checking Quantum Protocols. PhD thesis, University of Warwick (2009)
25. Selinger, P.: Towards a quantum programming language. Mathematical Structures in Computer Science 14(4), 527–586 (2004)
26. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Quantum Circuit Simulation. Springer (2009)
27. Ying, M., Feng, Y., Duan, R., Ji, Z.: An algebra of quantum processes. ACM Trans. Comput. Logic 10(3), 19:1–19:36 (2009)

Encoding Monomorphic and Polymorphic Types

Jasmin Christian Blanchette[1], Sascha Böhme[1],
Andrei Popescu[1], and Nicholas Smallbone[2]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Dept. of CSE, Chalmers University of Technology, Gothenburg, Sweden

Abstract. Most automatic theorem provers are restricted to untyped logics, and existing translations from typed logics are bulky or unsound. Recent research proposes monotonicity as a means to remove some clutter. Here we pursue this approach systematically, analysing formally a variety of encodings that further improve on efficiency while retaining soundness and completeness. We extend the approach to rank-1 polymorphism and present alternative schemes that lighten the translation of polymorphic symbols based on the novel notion of "cover". The new encodings are implemented, and partly proved correct, in Isabelle/HOL. Our evaluation finds them vastly superior to previous schemes.

1 Introduction

Specification languages, proof assistants, and other theorem proving applications typically rely on polymorphic types, but state-of-the-art automatic provers support only untyped or monomorphic logics. The existing sound and complete translation schemes for polymorphic types, whether they revolve around functions (tags) or predicates (guards), produce clutter that severely hampers the proof search, and lighter approaches based on type arguments are unsound [13, 15]. As a result, application authors face a difficult choice between soundness and efficiency.

The fourth author, together with Claessen and Lillieström [10], designed a pair of sound, complete, and efficient translations from monomorphic to untyped first-order logic with equality. The key insight is that *monotonic* types—types whose domain can be extended with new elements while preserving satisfiability—can be merged. The remaining types can be made monotonic by introducing protectors (tags or guards).

Example 1 (Monkey Village). Imagine a village of monkeys [10] where each monkey owns at least two bananas (b_1 and b_2):

$$\forall M\!:\!\mathit{monkey}.\ \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$$
$$\forall M\!:\!\mathit{monkey}.\ \mathsf{b}_1(M) \not\approx \mathsf{b}_2(M)$$
$$\forall M_1, M_2\!:\!\mathit{monkey},\ B\!:\!\mathit{banana}.\ \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 \approx M_2$$

The type *banana* is monotonic, whereas *monkey* is nonmonotonic because there can live at most $\lfloor b/2 \rfloor$ monkeys in a village with a finite supply of b bananas. Thanks to monotonicity, it is sound to omit all type information regarding *banana*s. The example can be encoded using a predicate $\mathsf{g}_{\mathit{monkey}}$ to guard against ill-typed *monkey* instantiations:

$$\forall M.\ \mathsf{g}_{\mathit{monkey}}(M) \rightarrow \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$$
$$\forall M.\ \mathsf{g}_{\mathit{monkey}}(M) \rightarrow \mathsf{b}_1(M) \not\approx \mathsf{b}_2(M)$$
$$\forall M_1, M_2, B.\ \mathsf{g}_{\mathit{monkey}}(M_1) \wedge \mathsf{g}_{\mathit{monkey}}(M_2) \wedge \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 \approx M_2$$

Monotonicity is not decidable, but it can often be inferred using suitable calculi. In this paper, we exploit this idea systematically, analysing a variety of encodings based on monotonicity: some are minor adaptations of existing ones, while others are novel encodings that further improve on the size of the translated formulas.

We also generalise the monotonicity approach to a rank-1 polymorphic logic, as embodied by TPTP TFF1 [3]. Unfortunately, the presence of a single equality literal $X^\alpha \approx t$ or $t \approx X^\alpha$, where X is a polymorphic variable of type α, will lead the analysis to classify all types as possibly nonmonotonic and force the use of protectors everywhere. We solve this issue through a novel scheme that reduces the clutter associated with nonmonotonic types, based on the observation that protectors are required only when translating the particular formulas that prevent a type from being inferred monotonic.

We first review four main traditional approaches (Sect. 2), which prepare the ground for the more advanced encodings. Next, we present known and novel monotonicity-based schemes that handle only ground types (Sect. 3); these are interesting in their own right and serve as stepping stones for the full-blown polymorphic encodings (Sect. 4). We also present alternative schemes that aim at reducing the clutter associated with polymorphic symbols, based on the novel notion of "cover" (Sect. 5). Proofs of correctness are included in a technical report [2].

A formalisation [4] of the results in the proof assistant Isabelle/HOL [14] is under way; it currently covers all the monomorphic encodings. The encodings have been implemented in Sledgehammer [13], which provides a bridge between Isabelle and automatic theorem provers. They were evaluated with E, iProver, SPASS, Vampire, and Z3 on a vast benchmark suite (Sect. 6).

2 Traditional Type Encodings

We assume that formulas are expressed in negation normal form (NNF), with negation applied to atoms, and that each variable is bound only once in a formula. Given a polymorphic signature $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ (with n-ary type constructors $\mathcal{K}$, function symbols $\mathcal{F}$, and predicate symbols $\mathcal{P}$, all three sets finite), symbols are declared as $\mathsf{s} : \forall\bar{\alpha}.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$, where ς is either a type (for $\mathsf{s} \in \mathcal{F}$) or o (for $\mathsf{s} \in \mathcal{P}$). An application $\mathsf{s}\langle\bar{\tau}\rangle(\bar{t})$ of s requires $|\bar{\alpha}|$ type arguments in angle brackets and $|\bar{\sigma}|$ term arguments in parentheses. We often omit $\langle\bar{\tau}\rangle$ in examples. Σ is *monomorphic* if none of the symbols take type arguments and *untyped* if additionally $\mathcal{K} = \{\iota\}$, in which case we omit $\mathcal{K}$ and indicate arities by superscripts (s^n). A *problem* over Σ is a finite set of closed formulas over Σ.

The easiest way to translate a typed problem into an untyped logic is to erase all type information, omitting type arguments, type quantifiers, and types in term quantifiers.

Definition 2 (Full Erasure e). The *full type erasure* encoding e translates a polymorphic problem over Σ into an untyped problem over Σ', where the symbols in Σ' have the same term arities as in Σ (but without type arguments). It is defined as follows:

$$\begin{array}{lll}
[\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathsf{e}} = \mathsf{f}([\![\bar{t}]\!]_{\mathsf{e}}) & [\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathsf{e}} = \mathsf{p}([\![\bar{t}]\!]_{\mathsf{e}}) & [\![\forall X{:}\sigma.\ \varphi]\!]_{\mathsf{e}} = \forall X.\ [\![\varphi]\!]_{\mathsf{e}} \\
[\![\forall\alpha.\ \varphi]\!]_{\mathsf{e}} = [\![\varphi]\!]_{\mathsf{e}} & [\![\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathsf{e}} = \neg\, \mathsf{p}([\![\bar{t}]\!]_{\mathsf{e}}) & [\![\exists X{:}\sigma.\ \varphi]\!]_{\mathsf{e}} = \exists X.\ [\![\varphi]\!]_{\mathsf{e}}
\end{array}$$

Here and elsewhere, we omit the trivial cases where the function is simply applied to its subterms or subformulas, as in $[\![\varphi_1 \wedge \varphi_2]\!]_{\mathsf{e}} = [\![\varphi_1]\!]_{\mathsf{e}} \wedge [\![\varphi_2]\!]_{\mathsf{e}}$.

By way of composition, the e encoding lies at the heart of all the encodings presented in this paper. Given n encodings $x_1, \ldots, x_n$, we write $[\![\ ]\!]_{x_1;\ldots;x_n}$ for $[\![\ ]\!]_{x_n} \circ \cdots \circ [\![\ ]\!]_{x_1}$.

Full type erasure is unsound in the presence of equality because equality can be used to encode cardinality constraints on domains. For example, $\forall U : \mathit{unit}.\ U \approx \mathsf{unity}$ forces the domain of *unit* to have only one element. Its erasure, $\forall U.\ U \approx \mathsf{unity}$, effectively restricts *all* types to one element. An additional issue is that erasure confuses distinct monomorphic instances of polymorphic symbols. The formula $\mathsf{q}\langle a\rangle(\mathsf{f}\langle a\rangle) \wedge \neg\, \mathsf{q}\langle b\rangle(\mathsf{f}\langle b\rangle)$ is satisfiable, but its type erasure $\mathsf{q}(\mathsf{f}) \wedge \neg\, \mathsf{q}(\mathsf{f})$ is unsatisfiable. A solution is to encode types as terms in the untyped logic: type variables α become term variables A, and type constructors k become function symbols k. A symbol with m type arguments is passed m additional term arguments. The example above is translated to $\mathsf{q}(\mathsf{a}, \mathsf{f}(\mathsf{a})) \wedge \neg\, \mathsf{q}(\mathsf{b}, \mathsf{f}(\mathsf{b}))$.

We call this encoding a. It coincides with e for monomorphic problems and is unsound. Nonetheless, it forms the basis of all our sound polymorphic encodings in a slightly generalised version, called a^x below. First, let us fix a distinguished type ϑ (for encoded types) and two symbols $\mathsf{t} : \forall\alpha.\ \alpha \to \alpha$ (for tags) and $\mathsf{g} : \forall\alpha.\ \alpha \to o$ (for guards).

Definition 3 (Type Argument Filter). Given a signature $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, a *type argument filter* x maps any $\mathsf{s} : \forall\alpha_1, \ldots, \alpha_m.\ \bar\sigma \to \varsigma$ to a subset $x_{\mathsf{s}} = \{i_1, \ldots, i_{m'}\} \subseteq \{1, \ldots, m\}$ of its type argument indices. Given a list $\bar z$ of length m, $x_{\mathsf{s}}(\bar z)$ denotes the sublist $z_{i_1}, \ldots, z_{i_{m'}}$, where $i_1 < \cdots < i_{m'}$. Filters are implicitly extended to $\{1\}$ for $\mathsf{t}, \mathsf{g} \notin \mathcal{F} \uplus \mathcal{P}$.

Definition 4 (Generic Arguments a^x). Given a type argument filter x, the *generic type arguments* encoding a^x translates a polymorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}, \mathcal{P}')$, where the symbols in $\mathcal{F}'$, $\mathcal{P}'$ are the same as those in $\mathcal{F}$, $\mathcal{P}$. For each symbol $\mathsf{s} : \forall\bar\alpha.\ \bar\sigma \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$, the arity of s in Σ' is $|x_{\mathsf{s}}| + |\bar\sigma|$. The encoding is defined as $[\![\ ]\!]_{\mathsf{a}^x;\mathsf{e}}$, where the non-trivial cases are

$$\begin{aligned}
[\![\mathsf{f}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathsf{a}^x} &= \mathsf{f}\langle\bar\sigma\rangle(\langle\!\langle x_{\mathsf{f}}(\bar\sigma)\rangle\!\rangle, [\![\bar t]\!]_{\mathsf{a}^x}) & [\![\forall\alpha.\ \varphi]\!]_{\mathsf{a}^x} &= \forall\alpha.\ \forall\langle\!\langle\alpha\rangle\!\rangle : \vartheta.\ [\![\varphi]\!]_{\mathsf{a}^x} \\
[\![\mathsf{p}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathsf{a}^x} &= \mathsf{p}\langle\bar\sigma\rangle(\langle\!\langle x_{\mathsf{p}}(\bar\sigma)\rangle\!\rangle, [\![\bar t]\!]_{\mathsf{a}^x}) & [\![\neg\, \mathsf{p}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathsf{a}^x} &= \neg\, [\![\mathsf{p}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathsf{a}^x}
\end{aligned}$$

The auxiliary function $\langle\!\langle\sigma\rangle\!\rangle$ returns the *term encoding* of a type over $\mathcal{K}$ as a term over $(\{\vartheta\}, \mathcal{K})$ of the distinguished type ϑ, following the simple scheme described above.

An intuitive approach to encode type information soundly is to wrap each term and subterm with its type using type tags. For polymorphic type systems, this scheme relies on a distinguished binary function $\mathsf{t}(\langle\!\langle\sigma\rangle\!\rangle, t)$ that "annotates" each term t with its type σ. The tags make most type arguments superfluous. This encoding is defined as a two-stage process: the first stage adds tags $\mathsf{t}\langle\sigma\rangle(t)$ while preserving the polymorphism; the second stage encodes t's type argument as well as any phantom type arguments.

Definition 5 (Phantom Type Argument). Let $\mathsf{s} : \forall\alpha_1, \ldots, \alpha_m.\ \bar\sigma \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. The ith type argument is a *phantom* if α_i does not occur in $\bar\sigma$ or ς. Given a list $\bar z \equiv z_1, \ldots, z_m$, $\mathsf{phan}_{\mathsf{s}}(\bar z)$ denotes the sublist $z_{i_1}, \ldots, z_{i_{m'}}$ corresponding to the phantom type arguments.

Definition 6 (Traditional Tags t). The *traditional type tags* encoding t translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K} \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{phan}}$ (i.e. a^x with $x = \mathsf{phan}$). It is defined as $[\![\ ]\!]_{\mathsf{t};\mathsf{a}^{\mathsf{phan}};\mathsf{e}}$, i.e. the composition of $[\![\ ]\!]_{\mathsf{t}}$, $[\![\ ]\!]_{\mathsf{a}^{\mathsf{phan}}}$ and $[\![\ ]\!]_{\mathsf{e}}$, where

$$[\![\mathsf{f}\langle\sigma\rangle(\bar t)]\!]_{\mathsf{t}} = \lfloor \mathsf{f}\langle\sigma\rangle([\![\bar t]\!]_{\mathsf{t}}) \rfloor \qquad [\![X]\!]_{\mathsf{t}} = \lfloor X \rfloor \qquad \text{with } \lfloor t^{\sigma} \rfloor = \mathsf{t}\langle\sigma\rangle(t)$$

Example 7 (Algebraic Lists). The following axioms induce a minimalistic first-order theory of algebraic lists that will serve as our main running example:

$$\forall\alpha.\ \forall X\!:\alpha,\ Xs\!:\mathit{list}(\alpha).\ \mathsf{nil} \not\approx \mathsf{cons}(X, Xs)$$
$$\forall\alpha.\ \forall X\!:\alpha,\ Xs\!:\mathit{list}(\alpha).\ \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X \wedge \mathsf{tl}(\mathsf{cons}(X, Xs)) \approx Xs$$

We conjecture that cons is injective. The conjecture's negation can be expressed employing an unknown but fixed Skolem type b:

$$\exists X, Y\!:b,\ Xs, Ys\!:\mathit{list}(b).\ \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Because the hd and tl equations force injectivity of cons in both arguments, the problem is unsatisfiable: the unnegated conjecture is a consequence of the axioms. The t encoding translates the problem into

$$\begin{aligned}
&\forall A, X, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{nil}) \not\approx \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs)))\\
&\forall A, X, Xs.\ \mathsf{t}(A, \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs))))) \approx \mathsf{t}(A, X) \wedge\\
&\qquad\qquad \mathsf{t}(\mathsf{list}(A), \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs))))) \approx \mathsf{t}(\mathsf{list}(A), Xs)\\
&\exists X, Y, Xs, Ys.\ \mathsf{t}(\mathsf{list}(\mathsf{b}), \mathsf{cons}(\mathsf{t}(\mathsf{b}, X), \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs))) \approx\\
&\qquad\qquad\quad \mathsf{t}(\mathsf{list}(\mathsf{b}), \mathsf{cons}(\mathsf{t}(\mathsf{b}, Y), \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys))) \wedge\\
&\qquad\qquad (\mathsf{t}(\mathsf{b}, X) \not\approx \mathsf{t}(\mathsf{b}, Y) \vee \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs) \not\approx \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys))
\end{aligned}$$

Type tags heavily burden the terms. An alternative is to introduce type guards, which are predicates that restrict the range of variables. They take the form of a distinguished predicate $\mathsf{g}(\langle\!\langle\sigma\rangle\!\rangle, t)$ that checks whether t has type σ. With the type tags encoding, only phantom type arguments needed to be encoded; here, we must encode any type arguments that cannot be read off the types of the term arguments. Thus, the type argument is encoded for $\mathsf{nil}\langle\alpha\rangle$ but omitted for $\mathsf{cons}\langle\alpha\rangle(X, Xs)$, $\mathsf{hd}\langle\alpha\rangle(Xs)$, and $\mathsf{tl}\langle\alpha\rangle(Xs)$.

Definition 8 (Inferable Type Argument). Let $\mathsf{s} : \forall\alpha_1,\ldots,\alpha_m.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. A type argument is *inferable* if it occurs in some of the term arguments' types. Given a list $\bar{z} \equiv z_1,\ldots,z_m$, $\mathsf{inf}_\mathsf{s}(\bar{z})$ denotes the sublist $z_{i_1},\ldots,z_{i_{m'}}$ corresponding to the inferable type arguments, and $\mathsf{ninf}_\mathsf{s}(\bar{z})$ denotes the sublist for noninferable type arguments.

Definition 9 (Traditional Guards g). The *traditional type guards* encoding g translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}, \mathcal{P}' \uplus \{\mathsf{g}^2\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$. It is defined as $[\![\ ]\!]_{\mathsf{g};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$, where

$$[\![\forall X\!:\sigma.\ \varphi]\!]_\mathsf{g} = \forall X\!:\sigma.\ \mathsf{g}\langle\sigma\rangle(X) \to [\![\varphi]\!]_\mathsf{g} \qquad [\![\exists X\!:\sigma.\ \varphi]\!]_\mathsf{g} = \exists X\!:\sigma.\ \mathsf{g}\langle\sigma\rangle(X) \wedge [\![\varphi]\!]_\mathsf{g}$$

The translation of a problem is given by $[\![\Phi]\!]_\mathsf{g} = \mathrm{Ax} \cup \bigcup_{\varphi\in\Phi} [\![\varphi]\!]_\mathsf{g}$, where Ax consists of the following *typing axioms*:

$$\forall\bar{\alpha}.\ \bar{X}\!:\bar{\sigma}.\ \big(\textstyle\bigwedge_j \mathsf{g}\langle\sigma_j\rangle(X_j)\big) \to \mathsf{g}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \qquad \text{for } \mathsf{f} : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F}$$
$$\forall\alpha.\ \exists X\!:\alpha.\ \mathsf{g}\langle\alpha\rangle(X)$$

The last axiom witnesses inhabitation of every type. It is necessary for completeness.

Example 10. The g encoding translates the algebraic list problem of Example 7 into

$$\begin{aligned}
&\forall A.\ \mathsf{g}(\mathsf{list}(A), \mathsf{nil}(A))\\
&\forall A, X, Xs.\ \mathsf{g}(A, X) \wedge \mathsf{g}(\mathsf{list}(A), Xs) \to \mathsf{g}(\mathsf{list}(A), \mathsf{cons}(X, Xs))\\
&\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs) \to \mathsf{g}(A, \mathsf{hd}(Xs))\\
&\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs) \to \mathsf{g}(\mathsf{list}(A), \mathsf{tl}(Xs))\\
&\forall A.\ \exists X.\ \mathsf{g}(A, X)
\end{aligned}$$

$$\forall A, X, Xs.\ \mathsf{g}(A, X) \wedge \mathsf{g}(\mathsf{list}(A), Xs) \rightarrow \mathsf{nil}(A) \not\approx \mathsf{cons}(X, Xs)$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \wedge \mathsf{g}(\mathsf{list}(A), Xs) \rightarrow \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X \wedge \mathsf{tl}(\mathsf{cons}(X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{g}(\mathsf{b}, X) \wedge \mathsf{g}(\mathsf{b}, Y) \wedge \mathsf{g}(\mathsf{list}(\mathsf{b}), Xs) \wedge \mathsf{g}(\mathsf{list}(\mathsf{b}), Ys) \wedge \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Bibliographical Notes. The earliest descriptions of type tags and type guards we are aware of are due to Enderton [11] and Stickel [15]. Wick and McCune [18] compare type arguments, tags, and guards in a monomorphic setting. Type arguments are reminiscent of System F; they are described by Meng and Paulson [13], who also consider full type erasure and polymorphic type tags. Urban [17] extended the untyped TPTP FOF syntax with dependent types to accommodate Mizar.

The intermediate verification language and tool Boogie 2 [12] supports a restricted form of higher-rank polymorphism (with polymorphic maps), and its cousin Why3 [6] provides rank-1 polymorphism. Both define translations to a monomorphic logic and handle interpreted types [7,12]. One of the Boogie translations [12] uses SMT triggers to prevent ill-typed instantiations. Bouillaguet et al. [8] showed that full type erasure is sound if all types can be assumed to have the same cardinality and exploit this in the verification system Jahob. An alternative to encoding polymorphic types is to support them natively in the prover; this is ubiquitous in interactive theorem provers, but perhaps the only automatic prover that supports polymorphism is Alt-Ergo [5].

3 Monotonicity-Based Type Encodings—The Monomorphic Case

Type tags and guards considerably increase the size of the problems passed to the automatic provers, with a dramatic impact on their performance. Most of the clutter can be removed by inferring monotonicity and soundly erasing type information based on the monotonicity analysis. Informally, a monotonic formula is one where, for any model of that formula, we can increase the size of the model while preserving satisfiability.

We focus on the monomorphic case, where the input problem contains no type variables or polymorphic symbols. Many of our definitions nonetheless handle the polymorphic case gracefully so that they can be reused in Section 4.

Before we start, let us define variants of the traditional t and g encodings that operate on monomorphic problems. The monomorphic encodings $\widetilde{\mathsf{t}}$ and $\widetilde{\mathsf{g}}$ coincide with t and g except that the polymorphic function $\mathsf{t}\langle\sigma\rangle(t)$ and predicate $\mathsf{g}\langle\sigma\rangle(t)$ are replaced by type-indexed families of unary functions $\mathsf{t}_\sigma(t)$ and predicates $\mathsf{g}_\sigma(t)$, as is customary in the literature [18].

Definition 11 (Monotonicity). Let S be a set of ground types and Φ be a problem. The types in S are (*infinitely*) *monotonic* in Φ if for all models $\mathcal{M}$ of Φ, there exists a model $\mathcal{M}'$ such that for all ground types σ, $[\![\sigma]\!]^{\mathcal{M}}$ is infinite if $\sigma \in S$ and $\left|[\![\sigma]\!]^{\mathcal{M}'}\right| = \left|[\![\sigma]\!]^{\mathcal{M}}\right|$ otherwise. A type σ is (*infinitely*) *monotonic* if $\{\sigma\}$ is monotonic. The problem Φ is (*infinitely*) *monotonic* if all its types, taken together, are monotonic.

Our criterion, infinite monotonicity, subsumes the finite monotonicity of Claessen et al. The set $\{\mathit{monkey}, \mathit{banana}\}$ is infinitely monotonic in Example 1, even though *banana* is not monotonic in the sense of Claessen et al. Another advantage of the new criterion is that it directly handles polymorphic signatures and infinitely many types.

Full type erasure is sound for monomorphic, monotonic problems. The intuition is that a model of such a problem can be extended into a model where all types are interpreted as sets of the same cardinality, which can be merged to yield an untyped model.

Claessen et al. introduced a simple calculus to infer finite monotonicity for monomorphic first-order logic [10]. The definition below generalises it from clause normal form to negation normal form. The calculus is based on the observation that a type σ must be monotonic if the problem expressed in NNF contains no positive literal of the form $X^\sigma \approx t$ or $t \approx X^\sigma$, where X is universal. We call such an occurrence of X a naked occurrence. Naked variables are the only way to express upper bounds on the cardinality of types in first-order logic.

Definition 12 (Naked Variable). The set of *naked variables* $\mathrm{NV}(\varphi)$ of a formula φ is defined as follows:

$$\begin{aligned}
\mathrm{NV}(\mathsf{p}\langle\bar\sigma\rangle(\bar t)) &= \emptyset & \mathrm{NV}(t_1 \approx t_2) &= \{t_1, t_2\} \cap \mathcal{V} \\
\mathrm{NV}(\neg\, \mathsf{p}\langle\bar\sigma\rangle(\bar t)) &= \emptyset & \mathrm{NV}(t_1 \not\approx t_2) &= \emptyset \\
\mathrm{NV}(\varphi_1 \wedge \varphi_2) &= \mathrm{NV}(\varphi_1) \cup \mathrm{NV}(\varphi_2) & \mathrm{NV}(\forall X\!:\!\sigma.\ \varphi) &= \mathrm{NV}(\varphi) \\
\mathrm{NV}(\varphi_1 \vee \varphi_2) &= \mathrm{NV}(\varphi_1) \cup \mathrm{NV}(\varphi_2) & \mathrm{NV}(\exists X\!:\!\sigma.\ \varphi) &= \mathrm{NV}(\varphi) - \{X\}
\end{aligned}$$

Variables of types other than σ are irrelevant when inferring whether σ is monotonic; a variable is problematic only if it occurs naked and has type σ. Annoyingly, a single naked variable of type σ will cause us to classify σ as possibly nonmonotonic.

We regain some precision by extending the calculus with an infinity analysis: trivially, all types with no finite models are monotonic. Abstracting over the specific analysis used to detect infinite types (e.g. Infinox [9]), we fix a set $\mathrm{Inf}(\Phi)$ of types whose interpretations are guaranteed to be infinite in all models of Φ. The monotonicity calculus takes $\mathrm{Inf}(\Phi)$ into account.

Definition 13 (Monotonicity Calculus $\rhd$). Let Φ be a monomorphic problem. A judgement $\sigma \rhd \varphi$ indicates that the ground type σ is inferred monotonic in $\varphi \in \Phi$. The *monotonicity calculus* consists of the following rules:

$$\frac{\sigma \in \mathrm{Inf}(\Phi)}{\sigma \rhd \varphi} \qquad \frac{\mathrm{NV}(\varphi) \cap \{X \mid X \text{ has type } \sigma\} = \emptyset}{\sigma \rhd \varphi}$$

Monotonic types can be soundly erased when translating from a monomorphic logic to an untyped logic. Nonmonotonic types in general cannot. Claessen et al. [10] point out that adding sufficiently many protectors to a nonmonotonic problem will make it monotonic, after which its types can be erased. Thus the following general two-stage procedure translates monomorphic problems to untyped first-order logic:

1. Introduce protectors (tags or guards) without erasing any types:
   (a) Introduce protectors for universal variables of possibly nonmonotonic types.
   (b) If necessary, generate *typing axioms* for any function symbol whose result type is possibly nonmonotonic, to make it possible to remove protectors.
2. Erase all the types.

The purpose of stage 1 is to make the problem monotonic while preserving satisfiability. This paves the way for the sound type erasure of stage 2.

The encoding $\widetilde{\mathsf{t}}?$, due to Claessen et al., specialises this procedure for tags. It is similar to the traditional encoding $\widetilde{\mathsf{t}}$ (the monomorphic t), except that it omits the tags for types that are inferred monotonic. By wrapping all naked variables (in fact, all terms) of possibly nonmonotonic types in a function term, stage 1 yields a monotonic problem.

Definition 14 (Lightweight Tags $\widetilde{\mathbf{t}}?$). The *monomorphic lightweight type tags* encoding $\widetilde{\mathsf{t}}?$ translates a monomorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathsf{t}^1_\sigma\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for e. It is defined as $[\![\;]\!]_{\widetilde{\mathsf{t}}?;\mathsf{e}}$, where

$$[\![\mathsf{f}(\bar{t})]\!]_{\widetilde{\mathsf{t}}?} = \lfloor \mathsf{f}([\![\bar{t}]\!]_{\widetilde{\mathsf{t}}?}) \rfloor \qquad [\![X]\!]_{\widetilde{\mathsf{t}}?} = \lfloor X \rfloor \qquad \text{with } \lfloor t^\sigma \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \\ \mathsf{t}_\sigma(t) & \text{otherwise} \end{cases}$$

Example 15. For a monomorphised version of Example 7, with α instantiated by b, the monomorphic type corresponding to $list(b)$ is monotonic by virtue of being infinite, whereas b cannot be inferred monotonic. The $\widetilde{\mathsf{t}}?$ encoding of the problem follows:

$$\begin{aligned}
&\forall X, Xs.\ \mathsf{nil}_b \not\approx \mathsf{cons}_b(\mathsf{t}_b(X), Xs)\\
&\forall X, Xs.\ \mathsf{t}_b(\mathsf{hd}_b(\mathsf{cons}_b(\mathsf{t}_b(X), Xs))) \approx \mathsf{t}_b(X) \wedge \mathsf{tl}_b(\mathsf{cons}_b(\mathsf{t}_b(X), Xs)) \approx Xs\\
&\exists X, Y, Xs, Ys.\ \mathsf{cons}_b(\mathsf{t}_b(X), Xs) \approx \mathsf{cons}_b(\mathsf{t}_b(Y), Ys) \wedge (\mathsf{t}_b(X) \not\approx \mathsf{t}_b(Y) \vee Xs \not\approx Ys)
\end{aligned}$$

The $\widetilde{\mathsf{t}}?$ encoding treats all variables of the same type uniformly. Hundreds of axioms can suffer because of one unhappy formula that uses a type nonmonotonically (or in a way that cannot be inferred monotonic). To address this, we introduce a lighter encoding: if a universal variable does not occur naked in a formula, its tag can safely be omitted.[1]

Our novel encoding $\widetilde{\mathsf{t}}??$ protects only naked variables and introduces equations $\mathsf{t}_\sigma(\mathsf{f}(\overline{X})^\sigma) \approx \mathsf{f}(\overline{X})$ to add or remove tags around each function symbol f whose result type σ is possibly nonmonotonic, and similarly for existential variables.

Definition 16 (Featherweight Tags $\widetilde{\mathbf{t}}??$). The *monomorphic featherweight type tags* encoding $\widetilde{\mathsf{t}}??$ translates a monomorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for $\widetilde{\mathsf{t}}?$. It is defined as $[\![\;]\!]_{\widetilde{\mathsf{t}}??;\mathsf{e}}$, where

$$[\![t_1 \approx t_2]\!]_{\widetilde{\mathsf{t}}??} = \lfloor [\![t_1]\!]_{\widetilde{\mathsf{t}}??} \rfloor \approx \lfloor [\![t_2]\!]_{\widetilde{\mathsf{t}}??} \rfloor$$

$$[\![\exists X\!:\!\sigma.\ \varphi]\!]_{\widetilde{\mathsf{t}}??} = \exists X\!:\!\sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathsf{t}}??} & \text{if } \sigma \rhd \Phi \\ \mathsf{t}_\sigma(X) \approx X \wedge [\![\varphi]\!]_{\widetilde{\mathsf{t}}??} & \text{otherwise} \end{cases}$$

with

$$\lfloor t^\sigma \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \text{ is not a universal variable} \\ \mathsf{t}_\sigma(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\begin{aligned}
&\forall \bar{X}\!:\!\bar{\sigma}.\ \mathsf{t}_\sigma(\mathsf{f}(\bar{X})) \approx \mathsf{f}(\bar{X}) && \text{for } \mathsf{f} : \bar{\sigma} \to \sigma \in \mathcal{F} \text{ such that } \sigma \not\rhd \Phi\\
&\exists X\!:\!\sigma.\ \mathsf{t}_\sigma(X) \approx X && \text{for } \sigma \not\rhd \Phi \text{ that is not the result type of a symbol in } \mathcal{F}
\end{aligned}$$

The side condition for the last axiom is a minor optimisation: it avoids asserting that σ is inhabited if the symbols in $\mathcal{F}$ already witness σ's inhabitation.

[1] This is related to the observation that only paramodulation from or into a variable can cause ill-typed instantiations in a resolution prover [18].

Example 17. The $\widetilde{\mathrm{t}}$?? encoding of Example 15 requires fewer tags than $\widetilde{\mathrm{t}}$?, at the cost of more type information (for hd and the existential variables of type b):

$$\begin{aligned}
&\forall Xs.\ \mathsf{t}_b(\mathsf{hd}_b(Xs)) \approx \mathsf{hd}_b(Xs)\\
&\forall X, Xs.\ \mathsf{nil}_b \not\approx \mathsf{cons}_b(X, Xs)\\
&\forall X, Xs.\ \mathsf{hd}_b(\mathsf{cons}_b(X, Xs)) \approx \mathsf{t}_b(X) \wedge \mathsf{tl}_b(\mathsf{cons}_b(X, Xs)) \approx Xs\\
&\exists X, Y, Xs, Ys.\ \mathsf{t}_b(X) \approx X \wedge \mathsf{t}_b(Y) \approx Y \wedge \mathsf{cons}_b(X, Xs) \approx \mathsf{cons}_b(Y, Ys) \wedge\\
&\qquad\qquad (X \not\approx Y \vee Xs \not\approx Ys)
\end{aligned}$$

The $\widetilde{\mathrm{g}}$? and $\widetilde{\mathrm{g}}$?? encodings are defined analogously to $\widetilde{\mathrm{t}}$? and $\widetilde{\mathrm{t}}$?? but using type guards. The $\widetilde{\mathrm{g}}$? encoding omits the guards for types that are inferred monotonic, whereas $\widetilde{\mathrm{g}}$?? omits more guards that are not needed to make the intermediate problem monotonic.

Definition 18 (Lightweight Guards $\widetilde{\mathbf{g}}$?). The *monomorphic lightweight type guards* encoding $\widetilde{\mathrm{g}}$? translates a monomorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^1_\sigma\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for e. It is defined as $[\![\ ]\!]_{\widetilde{\mathrm{g}}?;\mathsf{e}}$, where

$$[\![\forall X{:}\sigma.\ \varphi]\!]_{\widetilde{\mathrm{g}}?} = \forall X{:}\sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathrm{g}}?} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}_\sigma(X) \rightarrow [\![\varphi]\!]_{\widetilde{\mathrm{g}}?} & \text{otherwise} \end{cases}$$

$$[\![\exists X{:}\sigma.\ \varphi]\!]_{\widetilde{\mathrm{g}}?} = \exists X{:}\sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathrm{g}}?} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}_\sigma(X) \wedge [\![\varphi]\!]_{\widetilde{\mathrm{g}}?} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$\forall \bar{X}{:}\bar{\sigma}.\ \mathsf{g}_\sigma(\mathsf{f}(\bar{X}))$	for $\mathsf{f} : \bar{\sigma} \to \sigma \in \mathcal{F}$ such that $\sigma \not\rhd \Phi$
$\exists X : \sigma.\ \mathsf{g}_\sigma(X)$	for $\sigma \not\rhd \Phi$ that is not the result type of a symbol in $\mathcal{F}$

Example 19. The $\widetilde{\mathrm{g}}$? encoding of Example 15 is as follows:

$$\begin{aligned}
&\forall Xs.\ \mathsf{g}_b(\mathsf{hd}_b(Xs))\\
&\forall X,\ Xs.\ \mathsf{g}_b(X) \rightarrow \mathsf{nil}_b \not\approx \mathsf{cons}_b(X, Xs)\\
&\forall X{:}b,\ Xs.\ \mathsf{g}_b(X) \rightarrow \mathsf{hd}_b(\mathsf{cons}_b(X, Xs)) \approx X \wedge \mathsf{tl}_b(\mathsf{cons}_b(X, Xs)) \approx Xs\\
&\exists X,\ Y,\ Xs,\ Ys.\ \mathsf{g}_b(X) \wedge \mathsf{g}_b(Y) \wedge \mathsf{cons}_b(X, Xs) \approx \mathsf{cons}_b(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)
\end{aligned}$$

Our novel encoding $\widetilde{\mathrm{g}}$?? omits the guards for variables that do no occur naked, regardless of whether they are of a monotonic type.

Definition 20 (Featherweight Guards $\widetilde{\mathbf{g}}$??). The *monomorphic featherweight type guards* encoding $\widetilde{\mathrm{g}}$?? is identical to the lightweight encoding $\widetilde{\mathrm{g}}$? except that the condition "if $\sigma \rhd \Phi$" in the $\forall$ case is weakened to "if $\sigma \rhd \Phi$ or $X \notin \mathrm{NV}(\varphi)$".

Example 21. The $\widetilde{\mathrm{g}}$?? encoding of the algebraic list problem is identical to $\widetilde{\mathrm{g}}$? except that the $\mathsf{nil}_b \not\approx \mathsf{cons}_b$ axiom does not have any guard.

Theorem 22 (Soundness and Completeness). *Let Φ be a monomorphic problem, and let $x \in \{\widetilde{\mathrm{t}}?, \widetilde{\mathrm{t}}??, \widetilde{\mathrm{g}}?, \widetilde{\mathrm{g}}??\}$. The problems Φ and $[\![\Phi]\!]_{x;\mathsf{e}}$ are equisatisfiable.*

Section 4 will show how to translate polymorphic types soundly and completely. If we are willing to sacrifice completeness, an easy way to extend $\widetilde{\mathrm{t}}$?, $\widetilde{\mathrm{t}}$??, $\widetilde{\mathrm{g}}$?, and $\widetilde{\mathrm{g}}$?? to polymorphism is to perform *finite monomorphisation*: heuristically instantiate all type variables with suitable ground types, taking as many copies of the formulas as desired. Finite monomorphisation is generally incomplete [7], but by eliminating type variables it considerably simplifies the generated formulas, leading to very efficient encodings.

4 Complete Monotonicity-Based Encoding of Polymorphism

Finite monomorphisation is simple and effective, but its incompleteness can be a cause for worry, and its nonmodular nature makes it unsuitable for some applications that need to export an entire polymorphic theory independently of any conjecture. Here we adapt the monotonicity calculus and the monomorphic encodings to a polymorphic setting.

We start with a brief digression. With monotonicity-based encoding schemes, type arguments are needed to distinguish instances of polymorphic symbols. These additional arguments introduce clutter, which we can eliminate in some cases. The result is an optimised variant $\mathsf{a}^{\mathsf{ctor}}$ of the type arguments encoding a, which will serve as the foundation for $\mathtt{t?}$, $\mathtt{t??}$, $\mathtt{g?}$, and $\mathtt{g??}$. Consider a type $\mathsf{sum}(\alpha,\beta)$ that is axiomatised to be freely constructed by $\mathsf{inl} : \alpha \to \mathsf{sum}(\alpha,\beta)$ and $\mathsf{inr} : \beta \to \mathsf{sum}(\alpha,\beta)$. Regardless of β, inl must be interpreted as an injection from α to $\mathsf{sum}(\alpha,\beta)$. For a fixed α, its interpretations for different β instances are isomorphic. As a result, it is safe to omit the type argument for β when encoding $\mathsf{inl}\langle\alpha,\beta\rangle$ and that for α in $\mathsf{inr}\langle\alpha,\beta\rangle$ and $\mathsf{nil}\langle\alpha\rangle : \mathit{list}(\alpha)$. In general, the type arguments that can be omitted for constructors are precisely those that are noninferable in the sense of Definition 8. We call this encoding $\mathsf{a}^{\mathsf{ctor}}$. The encodings presented below exploit the fact that $[\![\Phi]\!]_{\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ is equisatisfiable to Φ if Φ is monotonic.

The polymorphic version of the monotonicity calculus captures the insight that a polymorphic type is monotonic if each of its common instances with the type of any naked variable is an instance of an infinite type.

Definition 23 (Monotonicity Calculus $\rhd$). Let Φ be a polymorphic problem. The *monotonicity calculus* consists of the single rule

$$\frac{\forall X^{\tau} \in \mathrm{NV}(\varphi).\ \mathrm{mgu}(\sigma,\tau) \in \mathrm{Inf}^{*}(\varphi)}{\sigma \rhd \varphi}$$

where $\mathrm{mgu}(\sigma,\tau)$ is the most general unifier of σ and τ, and $\mathrm{Inf}^{*}(\varphi)$ consists of all instances of all types in $\mathrm{Inf}(\varphi)$.

The polymorphic $\mathtt{t?}$ encoding can be seen as a hybrid between traditional tags ($\mathtt{t}$) and monomorphic lightweight tags ($\widetilde{\mathtt{t?}}$): as in $\mathtt{t}$, tags take the form of a function $\mathsf{t}\langle\sigma\rangle(t)$; as in $\widetilde{\mathtt{t?}}$, tags are omitted for types that are inferred monotonic.

The main novelty concerns the typing axioms. The $\widetilde{\mathtt{t?}}$ encoding omits all typing axioms for infinite types. In the polymorphic case, the infinite type σ might be an instance of a more general, potentially finite type for which tags are generated. For example, if α is tagged (because it is possibly nonmonotonic) but its instance $\mathit{list}(\alpha)$ is not (because it is infinite), there will be mismatches between tagged and untagged terms. Our solution is to add the typing axiom $\mathsf{t}\langle\mathit{list}(\alpha)\rangle(\mathit{Xs}) \approx \mathit{Xs}$, which allows the prover to add or remove a tag for the infinite type $\mathit{list}(\alpha)$. Such an axiom is sound for any monotonic type.

Definition 24 (Lightweight Tags $\mathtt{t?}$). The *polymorphic lightweight type tags* encoding $\mathtt{t?}$ translates a polymorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ctor}}$. It is defined as $[\![\ ]\!]_{\mathtt{t?};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![\mathsf{f}\langle\sigma\rangle(\bar{t})^{\sigma}]\!]_{\mathtt{t?}} = \lfloor \mathsf{f}\langle\sigma\rangle([\![\bar{t}]\!]_{\mathtt{t?}})\rfloor \quad [\![X^{\sigma}]\!]_{\mathtt{t?}} = \lfloor X \rfloor \quad \text{with } \lfloor t^{\sigma} \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \\ \mathsf{t}\langle\sigma\rangle(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by the following typing axioms, where ρ is a type substitution and $\mathrm{TV}(\sigma\rho)$ denotes the type variables of $\sigma\rho$:

$$\forall \mathrm{TV}(\sigma\rho).\ \forall X{:}\,\sigma\rho.\ \mathsf{t}\langle\sigma\rho\rangle(X) \approx X \qquad \text{for } \sigma\rho \in \mathrm{Inf}(\Phi) \text{ such that } \sigma \not\rhd \Phi$$

The lighter encoding $\mathsf{t}??$ protects only naked variables and introduces equations of the form $\mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\overline{X})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\overline{X})$ to add or remove tags around each function symbol f of a possibly nonmonotonic type σ, and similarly for existential variables.

Definition 25 (Featherweight Tags $\mathsf{t}??$). The *polymorphic featherweight type tags* encoding $\mathsf{t}??$ translates a polymorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for $\mathsf{t}?$. It is defined as $[\![\ ]\!]_{\mathsf{t}??;\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![t_1 \approx t_2]\!]_{\mathsf{t}??} = \lfloor [\![t_1]\!]_{\mathsf{t}??} \rfloor \approx \lfloor [\![t_2]\!]_{\mathsf{t}??} \rfloor$$
$$[\![\exists X{:}\,\sigma.\ \varphi]\!]_{\mathsf{t}??} = \exists X{:}\,\sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{t}??} & \text{if } \sigma \rhd \Phi \\ \mathsf{t}\langle\sigma\rangle(X) \approx X \wedge [\![\varphi]\!]_{\mathsf{t}??} & \text{otherwise} \end{cases}$$

with

$$\lfloor t^{\sigma} \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \text{ is not a universal variable} \\ \mathsf{t}\langle\sigma\rangle(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$\forall\bar{\alpha}.\ \forall\bar{X}{:}\,\bar{\sigma}.\ \mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})$	for $\mathsf{f} : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F}$ such that $\exists\rho.\ \sigma\rho \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma\rho).\ \forall X{:}\,\sigma\rho.\ \mathsf{t}\langle\sigma\rho\rangle(X) \approx X$	for $\sigma\rho \in \mathrm{Inf}(\Phi)$ such that $\sigma \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma).\ \exists X{:}\,\sigma.\ \mathsf{t}\langle\sigma\rangle(X) \approx X$	for $\sigma \not\rhd \Phi$ that is not an instance of the result type of $\mathsf{f} \in \mathcal{F}$ or a proper instance of $\tau \not\rhd \Phi$

Example 26. In Example 7, *list*(α) is infinite and hence monotonic, whereas α and its instance b cannot be inferred monotonic. The $\mathsf{t}??$ encoding of the problem follows:

$$\forall A, Xs.\ \mathsf{t}(A, \mathsf{hd}(A, Xs)) \approx \mathsf{hd}(A, Xs)$$
$$\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), Xs) \approx Xs$$
$$\forall A.\ \exists X.\ \mathsf{t}(A, X) \approx X$$
$$\forall A, X, Xs.\ \mathsf{nil} \not\approx \mathsf{cons}(A, X, Xs)$$
$$\forall A, X, Xs.\ \mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) \approx \mathsf{t}(A, X) \wedge \mathsf{tl}(A, \mathsf{cons}(A, X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{t}(\mathsf{b}, X) \approx X \wedge \mathsf{t}(\mathsf{b}, Y) \approx Y \wedge \mathsf{cons}(\mathsf{b}, X, Xs) \approx \mathsf{cons}(\mathsf{b}, Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Analogously to $\mathsf{t}?$, the $\mathsf{g}?$ encoding is best understood as a hybrid between traditional guards (g) and monomorphic lightweight guards ($\tilde{\mathsf{g}}?$): as in g, guards take the form of a predicate $\mathsf{g}\langle\sigma\rangle(t)$; as in $\tilde{\mathsf{g}}?$, guards are omitted for types that are inferred monotonic.

Once again, the main novelty concerns the typing axioms. The $\tilde{\mathsf{g}}?$ encoding omits all typing axioms for infinite types. In the polymorphic case, the infinite type σ might be an instance of a more general, potentially finite type for which guards are generated. Our solution is to add the typing axiom $\mathsf{g}\langle\sigma\rangle(X)$, which allows the prover to discharge any guard for the infinite type σ.

Definition 27 (Lightweight Guards g?). The *polymorphic lightweight type guards* encoding g? translates a polymorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^2\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ctor}}$. It is defined as $[\![\ ]\!]_{\mathsf{g?};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![\forall X\!:\!\sigma.\ \varphi]\!]_{\mathsf{g?}} = \forall X\!:\!\sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{g?}} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}\langle\sigma\rangle(X) \rightarrow [\![\varphi]\!]_{\mathsf{g?}} & \text{otherwise} \end{cases}$$

$$[\![\exists X\!:\!\sigma.\ \varphi]\!]_{\mathsf{g?}} = \exists X\!:\!\sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{g?}} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}\langle\sigma\rangle(X) \wedge [\![\varphi]\!]_{\mathsf{g?}} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$\forall\bar\alpha.\ \forall\bar X\!:\!\bar\sigma.\ \mathsf{g}\langle\sigma\rangle(\mathsf{f}\langle\bar\alpha\rangle(\bar X))$	for $\mathsf{f} : \forall\bar\alpha.\ \bar\sigma \to \sigma \in \mathcal{F}$ such that $\exists\rho.\ \sigma\rho \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma\rho).\ \forall X\!:\!\bar\sigma\rho.\ \mathsf{g}\langle\sigma\rho\rangle(X)$	for $\sigma\rho \in \mathrm{Inf}(\Phi)$ such that $\sigma \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma).\ \exists X\!:\!\sigma.\ \mathsf{g}\langle\sigma\rangle(X)$	for $\sigma \not\rhd \Phi$ that is not an instance of the result type of $\mathsf{f} \in \mathcal{F}$ or a proper instance of $\tau \not\rhd \Phi$

The featherweight cousin is a straightforward generalisation of g?.

Definition 28 (Featherweight Guards g??). The *polymorphic featherweight type guards* encoding g?? is identical to the lightweight encoding g? except that the condition "if $\sigma \rhd \Phi$" in the $\forall$ case is weakened to "if $\sigma \rhd \Phi$ or $X \notin \mathrm{NV}(\varphi)$".

Example 29. The g?? encoding of Example 7 follows:

$$\forall A, Xs.\ \mathsf{g}(A, \mathsf{hd}(A, Xs))$$
$$\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs)$$
$$\forall A, X, Xs.\ \mathsf{nil} \not\approx \mathsf{cons}(A, X, Xs)$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \rightarrow \mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) \approx X \wedge \mathsf{tl}(A, \mathsf{cons}(A, X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{g}(\mathsf{b}, X) \wedge \mathsf{g}(\mathsf{b}, Y) \wedge \mathsf{cons}(\mathsf{b}, X, Xs) \approx \mathsf{cons}(\mathsf{b}, Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Theorem 30 (Soundness and Completeness). *Let Φ be a polymorphic problem, and let $x \in \{\mathsf{t?}, \mathsf{t??}, \mathsf{g?}, \mathsf{g??}\}$. The problems Φ and $[\![\Phi]\!]_{x;\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ are equisatisfiable.*

5 Alternative, Cover-Based Encoding of Polymorphism

An issue with t?, t??, g?, and g?? is that they clutter the generated problem with type arguments. In that respect, the traditional t and g encodings are superior—t omits all non-phantom type arguments, and g omits all inferable type arguments. This would be unsound for the monotonicity-based encodings, because these leave out many of the protectors that implicitly "carry", or "cover", the type arguments in the traditional encodings. Nonetheless, an alternative is possible: by keeping more protectors around, we can omit inferable type arguments.

Definition 31 (Cover). Let $\mathsf{s} : \forall\bar\alpha.\ \bar\sigma \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. A (*type argument*) *cover* $C \subseteq \{1, \ldots, |\bar\sigma|\}$ for s is a set of term argument indices such that any inferable type argument can be inferred from a term argument whose index is in C. We let $\mathrm{Cover}_{\mathsf{s}}$ denote an arbitrary but fixed minimal cover of s.

For example, $\{1\}$ and $\{2\}$ are minimal covers for $\mathsf{cons} : \forall\alpha.\ \alpha \times \mathit{list}(\alpha) \to \mathit{list}(\alpha)$, and $\{1,2\}$ is also a cover. As canonical cover, we arbitrarily choose $\mathrm{Cover}_{\mathsf{cons}} = \{1\}$.

The encodings $\mathsf{t@}$ and $\mathsf{g@}$ introduced below ensure that each argument that is part of its enclosing function or predicate's cover has a unique type, from which the omitted type arguments can be inferred. For example, $\mathsf{t@}$ translates the term $\mathsf{cons}\langle\alpha\rangle(X, Xs)$ to $\mathsf{cons}(\mathsf{t}(A, X), Xs)$ with a type tag around X, effectively preventing an ill-typed instantiation of X that would result in the wrong type argument being inferred. We call variables that occur in their enclosing symbol's cover "undercover variables". They can be seen as a generalisation of naked variables to arbitrary predicate and function symbols.

Definition 32 (Undercover Variable). The set of *undercover variables* $\mathrm{UV}(\varphi)$ of a formula φ is defined by the equations

$$\begin{aligned}
\mathrm{UV}(\mathsf{f}\langle\bar\sigma\rangle(\bar t)) &= \lfloor\bar t\rfloor_{\mathsf{f}} \cup \mathrm{UV}(\bar t) & \mathrm{UV}(X) &= \emptyset\\
\mathrm{UV}(\mathsf{p}\langle\bar\sigma\rangle(\bar t)) &= \lfloor\bar t\rfloor_{\mathsf{p}} \cup \mathrm{UV}(\bar t) & \mathrm{UV}(t_1 \approx t_2) &= (\{t_1, t_2\} \cap \mathcal{V}) \cup \mathrm{UV}(t_1, t_2)\\
\mathrm{UV}(\neg\,\mathsf{p}\langle\bar\sigma\rangle(\bar t)) &= \lfloor\bar t\rfloor_{\mathsf{p}} \cup \mathrm{UV}(\bar t) & \mathrm{UV}(t_1 \not\approx t_2) &= \mathrm{UV}(t_1, t_2)\\
\mathrm{UV}(\varphi_1 \wedge \varphi_2) &= \mathrm{UV}(\varphi_1, \varphi_2) & \mathrm{UV}(\forall X{:}\,\sigma.\ \varphi) &= \mathrm{UV}(\varphi)\\
\mathrm{UV}(\varphi_1 \vee \varphi_2) &= \mathrm{UV}(\varphi_1, \varphi_2) & \mathrm{UV}(\exists X{:}\,\sigma.\ \varphi) &= \mathrm{UV}(\varphi) - \{X\}
\end{aligned}$$

where $\lfloor\bar t\rfloor_{\mathsf{s}} = \{t_j \mid j \in \mathrm{Cover}_{\mathsf{s}}\} \cap \mathcal{V}$ and $\mathrm{UV}(\bar t) = \bigcup_j \mathrm{UV}(t_j)$.

The cover-based encoding $\mathsf{t@}$ is similar to the traditional encoding t, except that it tags only undercover occurrences of variables and requires typing axioms.

Definition 33 (Cover Tags $\mathsf{t@}$). The *polymorphic cover-based type tags* encoding $\mathsf{t@}$ translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K} \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$. It is defined as $[\![\ ]\!]_{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$, where

$$\begin{aligned}
[\![\mathsf{f}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathsf{t@}} &= \mathsf{f}\langle\bar\sigma\rangle(\lfloor[\![\bar t]\!]_{\mathsf{t@}}\rfloor_{\mathsf{f}}) & [\![t_1 \approx t_2]\!]_{\mathsf{t@}} &= \lfloor[\![t_1]\!]_{\mathsf{t@}}\rfloor_{\approx} \approx \lfloor[\![t_2]\!]_{\mathsf{t@}}\rfloor_{\approx}\\
[\![\mathsf{p}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathsf{t@}} &= \mathsf{p}\langle\bar\sigma\rangle(\lfloor[\![\bar t]\!]_{\mathsf{t@}}\rfloor_{\mathsf{p}}) & [\![\exists X{:}\,\sigma.\ \varphi]\!]_{\mathsf{t@}} &= \exists X{:}\,\sigma.\ \mathsf{t}\langle\sigma\rangle(X) \approx X \wedge [\![\varphi]\!]_{\mathsf{t@}}\\
[\![\neg\,\mathsf{p}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathsf{t@}} &= \neg\,\mathsf{p}\langle\bar\sigma\rangle(\lfloor[\![\bar t]\!]_{\mathsf{t@}}\rfloor_{\mathsf{p}})
\end{aligned}$$

The auxiliary function $\lfloor(t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})\rfloor_{\mathsf{s}}$ returns a vector $(u_1, \ldots, u_n)$ such that

$$u_j = \begin{cases} t_j & \text{if } j \notin \mathrm{Cover}_{\mathsf{s}} \text{ or } t_j \text{ is not a universal variable}\\ \mathsf{t}\langle\sigma_j\rangle(t_j) & \text{otherwise}\end{cases}$$

taking $\mathrm{Cover}_{\approx} = \{1, 2\}$. The encoding is complemented by typing axioms:

$$\begin{aligned}
&\forall\bar\alpha.\ \forall\bar X{:}\,\bar\sigma.\ \mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar\alpha\rangle(\lfloor\bar X\rfloor_{\mathsf{f}})) \approx \mathsf{f}\langle\bar\alpha\rangle(\lfloor\bar X\rfloor_{\mathsf{f}}) \qquad \text{for } \mathsf{f} : \forall\bar\alpha.\ \bar\sigma \to \sigma \in \mathcal{F}\\
&\forall\alpha.\ \exists X{:}\,\alpha.\ \mathsf{t}\langle\alpha\rangle(X) \approx X
\end{aligned}$$

Example 34. The $\mathsf{t@}$ encoding of Example 7 is as follows:

$$\begin{aligned}
&\forall A.\ \mathsf{t}(\mathsf{list}(A), \mathsf{nil}(A)) \approx \mathsf{nil}(A)\\
&\forall A, X, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{cons}(\mathsf{t}(A, X), Xs)\\
&\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))\\
&\forall A, Xs.\ \mathsf{t}(A, \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))\\
&\forall A, X, Xs.\ \mathsf{nil}(A) \not\approx \mathsf{cons}(\mathsf{t}(A, X), Xs)\\
&\forall A, X, Xs.\ \mathsf{hd}(\mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{t}(A, X) \wedge \mathsf{tl}(\mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{t}(\mathsf{list}(A), Xs)\\
&\exists X, Y, Xs, Ys.\ \mathsf{t}(\mathsf{b}, X) \approx X \wedge \mathsf{t}(\mathsf{b}, Y) \approx Y \wedge \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs) \approx Xs \wedge \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys) \approx Ys \wedge\\
&\qquad\qquad \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)
\end{aligned}$$

Definition 35 (Cover Guards $\mathsf{g@}$). The *polymorphic cover-based type guards* encoding $\mathsf{g@}$ is identical to the traditional g encoding except for the $\forall$ case:

$$[\![\forall X : \sigma.\ \varphi]\!]_{\mathsf{g@}} = \forall X : \sigma. \begin{cases} [\![\varphi]\!]_{\mathsf{g@}} & \text{if } X \notin \mathrm{UV}(\varphi) \\ \mathsf{g}\langle\sigma\rangle(X) \rightarrow [\![\varphi]\!]_{\mathsf{g@}} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\forall\bar{\alpha}.\ \bar{X} : \bar{\sigma}.\ \big(\textstyle\bigwedge_{j \in \mathrm{Cover}_{\mathsf{f}}} \mathsf{g}\langle\sigma_j\rangle(X_j)\big) \rightarrow \mathsf{g}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \qquad \text{for } \mathsf{f} : \forall\bar{\alpha}.\ \bar{\sigma} \rightarrow \sigma \in \mathcal{F}$$
$$\forall\alpha.\ \exists X : \alpha.\ \mathsf{g}\langle\alpha\rangle(X)$$

Example 36. The $\mathsf{g@}$ encoding of the algebraic list problem is identical to the g encoding (Example 10), except that the guard on Xs is omitted in two of the axioms:

$$\forall A, X, Xs.\ \mathsf{g}(A, X) \rightarrow \mathsf{g}(\mathsf{list}(A), \mathsf{cons}(X, Xs))$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \rightarrow \mathsf{nil}(A) \not\approx \mathsf{cons}(X, Xs)$$

Theorem 37 (Soundness and Completeness). *Let Φ be a polymorphic problem, and let $\bar{x} \in \{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}}, \mathsf{g@};\mathsf{a}^{\mathsf{phan}}\}$. The problems Φ and $[\![\Phi]\!]_{\bar{x};\mathsf{e}}$ are equisatisfiable.*

6 Evaluation

To evaluate the type encodings described in this paper, we put together a set of 1000 polymorphic first-order problems originating from 10 existing Isabelle theories, translated with Sledgehammer's help. Our test data are publicly available [1].

The problems include up to 500 heuristically selected facts. We evaluated each type encoding with five modern automatic provers: E 1.6, iProver 0.99, SPASS 3.8ds, Vampire 2.6, and Z3 4.0. To make the evaluation more informative, we also tested the provers' native support for monomorphic types where it is available; it is referred to as $\widetilde{\mathsf{n}}$. Each prover was invoked with the set of options we had previously determined worked best for Sledgehammer.[2] The provers were granted 20 seconds of CPU time per problem on one core of a 3.06 GHz Dual-Core Intel Xeon processor. To avoid giving the unsound encodings an unfair advantage, for these proof search was followed by a certification phase that attempted to re-find the proof using a combination of sound encodings, based on its referenced facts. This phase slightly penalises the unsound encodings by rejecting a few sound proofs, but such is the price of unsoundness.

Figure 1 gives, for each combination of prover and encoding, the number of solved problems. Rows marked with $\sim$ concern the monomorphic encodings. The encodings $\widetilde{\mathsf{a}}$, $\widetilde{\mathsf{a}}^{\mathsf{ctor}}$, $\widetilde{\mathsf{t@}}$, and $\widetilde{\mathsf{g@}}$ are omitted; the first two coincide with $\widetilde{\mathsf{e}}$, whereas $\widetilde{\mathsf{t@}}$ and $\widetilde{\mathsf{g@}}$ are identical to versions of $\widetilde{\mathsf{t}}\mathsf{??}$ and $\widetilde{\mathsf{g}}\mathsf{??}$ that treat all types as possibly nonmonotonic. Among the encodings to untyped first-order logic, the monomorphic featherweight encoding $\widetilde{\mathsf{g}}\mathsf{??}$ performed best overall. It even outperformed Vampire's recently added native types ($\widetilde{\mathsf{n}}$). Among the polymorphic encodings, our novel monotonicity-based and cover-based encodings ($\mathsf{t?}$, $\mathsf{t??}$, $\mathsf{t@}$, $\mathsf{g?}$, $\mathsf{g??}$, and $\mathsf{g@}$), with the exception of $\mathsf{t@}$, constitute a substantial improvement over the traditional sound schemes (t and g).

[2] The setup for E was suggested by Stephan Schulz and includes the little known "symbol offset" weight function. We ran iProver with the default setup, SPASS in Isabelle mode, Vampire in CASC mode, and Z3 in TPTP mode with model-based quantifier instantiation enabled.

		e	a	t	t?	t??	t@	g	g?	g??	g@	n
E		116	**361**	263	275	347	228	216	344	349	262	–
	~	393	–	328	390	397	–	337	393	**401**	–	–
iProver		243	212	231	202	**262**	135	140	242	257	169	–
	~	210	–	243	246	245	–	180	**247**	241	–	–
SPASS		131	292	262	245	**299**	164	164	283	296	208	–
	~	331	–	293	326	330	–	237	320	334	–	**356**
Vampire		120	**341**	277	281	314	212	171	271	299	241	–
	~	393	–	309	379	382	–	265	390	**403**	–	372
Z3		281	**355**	250	238	350	279	213	291	351	268	–
	~	354	–	268	343	346	–	328	**355**	349	–	350

Fig. 1. Number of solved problems

The new type encodings also made an impact at the 2012 edition of CASC, the annual automatic prover competition [16]. Isabelle competes against LEO-II, Satallax, and TPS in the higher-order division. Largely thanks to the new schemes (but also to improvements in the underlying first-order provers), Isabelle moved from the third place it had occupied since 2009 to the first place.

7 Conclusion

This paper introduced a family of translations from polymorphic into untyped first-order logic, with a focus on efficiency. Our monotonicity-based encodings soundly erase all types that are inferred monotonic, as well as most occurrences of the remaining types. The best translations outperform the traditional encoding schemes.

We implemented the new translations in the Sledgehammer tool for Isabelle/HOL and the companion proof method *metis*, thereby addressing a recurring user complaint. Although Isabelle certifies external proofs, unsound proofs are annoying and often conceal sound proofs. The same translation module forms the core of Isabelle's TPTP exporter tool, which makes entire theorem libraries available to first-order reasoners. Our refinements to the monomorphic case have made their way into Monotonox [10]. Applications such as Boogie [12] and Why3 [6] also stand to gain from lighter encodings.

The TPTP family recently welcomed the addition of TFF1 [3], an extension of the monomorphic TFF0 logic with rank-1 polymorphism. Equipped with a concrete syntax and translation tools, we can turn any popular automatic theorem prover into an efficient polymorphic prover. Translating the untyped proof back into a typed proof is usually straightforward, but there are important corner cases that call for more research.

The encodings are all instances of a general framework, in which mostly orthogonal features can be combined in various ways. Defining such a large number of encodings makes it possible to select the most appropriate scheme for each automatic prover, based on empirical evidence. In fact, using time slicing or parallelism, it pays off to have each prover employ a combination of encodings with complementary strengths.

Acknowledgement. Koen Claessen and Tobias Nipkow made this collaboration possible. Lukas Bulwahn, Peter Lammich, Rustan Leino, Tobias Nipkow, Mark Summerfield, Tjark Weber, and several anonymous reviewers suggested dozens of textual improvements. We thank them all. The first author's research was supported by the Deutsche Forschungsgemeinschaft (grants Ni 491/11-2 and Ni 491/14-1). The authors are listed in alphabetical order regardless of individual contributions or seniority.

References

[1] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Empirical data associated with this paper (2012), http://www21.in.tum.de/~blanchet/enc_types_data.tar.gz

[2] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. Tech. report (2012), http://www21.in.tum.de/~blanchet/enc_types_report.pdf

[3] Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism. Tech. report (2012), http://www21.in.tum.de/~blanchet/tff1spec.pdf

[4] Blanchette, J.C., Popescu, A.: Formal development associated with this paper (2012), http://www21.in.tum.de/~popescua/enc_types_devel.zip

[5] Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (eds.) SMT 2008 (2008)

[6] Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Leino, K.R.M., Moskal, M. (eds.) Boogie 2011, pp. 53–64 (2011)

[7] Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 87–102. Springer, Heidelberg (2011)

[8] Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.: Using First-Order Theorem Provers in the Jahob Data Structure Verification System. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 74–88. Springer, Heidelberg (2007)

[9] Claessen, K., Lillieström, A.: Automated inference of finite unsatisfiability. J. Autom. Reasoning 47(2), 111–132 (2011)

[10] Claessen, K., Lillieström, A., Smallbone, N.: Sort It Out with Monotonicity: Translating between Many-Sorted and Unsorted First-Order Logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 207–221. Springer, Heidelberg (2011)

[11] Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)

[12] 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)

[13] Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reasoning 40(1), 35–60 (2008)

[14] Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)

[15] Stickel, M.E.: Schubert's steamroller problem: Formulations and solutions. J. Autom. Reasoning 2(1), 89–101 (1986)

[16] Sutcliffe, G.: Proceedings of the 6th IJCAR ATP system competition (CASC-J6). In: Sutcliffe, G. (ed.) CASC-J6. EPiC, vol. 11, pp. 1–50. EasyChair (2012)

[17] Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. J. Autom. Reasoning 37(1-2), 21–43 (2006)

[18] Wick, C.A., McCune, W.W.: Automated reasoning about elementary point-set topology. J. Autom. Reasoning 5(2), 239–255 (1989)

Deriving Probability Density Functions from Probabilistic Functional Programs

Sooraj Bhat[1], Johannes Borgström[2], Andrew D. Gordon[3], and Claudio Russo[3]

[1] Georgia Institute of Technology
[2] Uppsala University
[3] Microsoft Research

Abstract. The *probability density function* of a probability distribution is a fundamental concept in probability theory and a key ingredient in various widely used machine learning methods. However, the necessary framework for compiling probabilistic functional programs to density functions has only recently been developed. In this work, we present a density compiler for a probabilistic language with discrete and continuous distributions, and discrete observations, and provide a proof of its soundness. The compiler greatly reduces the development effort of domain experts, which we demonstrate by solving inference problems from various scientific applications, such as modelling the global carbon cycle, using a standard Markov chain Monte Carlo framework.

1 Introduction

Probabilistic programming promises to arm data scientists with declarative languages for specifying their probabilistic models, while leaving the details of how to translate those models to efficient sampling or inference algorithms to a compiler. Many widely used machine learning techniques that might be employed by such a compiler use as input the *probability density function* (PDF) of the model. Such techniques include *maximum likelihood* or *maximum a posteriori estimation*, *L2 estimation*, *importance sampling*, and *Markov chain Monte Carlo* (MCMC) methods.

Despite their utility, density functions have been largely absent from the literature on probabilistic functional programming. This is because the relationship between programs and their density functions is not straightforward: for a given program, the PDF may not exist or may be non-trivial to calculate. Such programs are not merely infrequent pathological curiosities but in fact arise in many ordinary scenarios. In this paper, we define, prove correct, and implement an algorithm for automatically computing PDFs for a large class of programs written in a rich probabilistic programming language.

Probability Density Functions. We now explain what a probability density function is, where it arises, and what we use it for in this paper. Consider a probabilistic program that generates outcomes from a set Ω. The *probability distribution* $\mathbb{P}$ of the program characterizes its behavior by assigning probabilities to different *subsets (events)* of Ω, denoting the proportion of runs that generate an outcome in that subset.

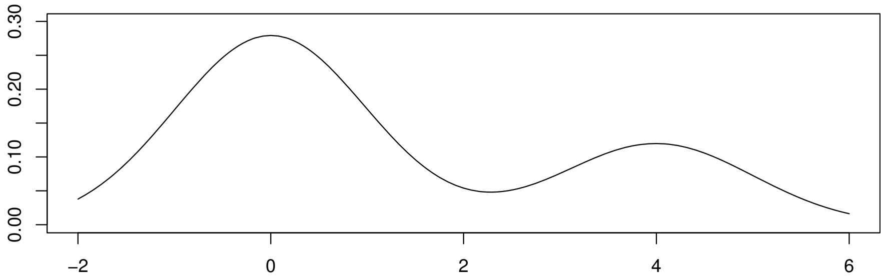
It turns out to be more productive to work with a function on the *elements* of Ω instead of the *subsets* of Ω, which characterizes the distribution. There is always such a function when Ω is countable, known as the *probability mass function*. It is defined $f(x) \triangleq \mathbb{P}(\{x\})$ and enjoys the property $\mathbb{P}(A) = \sum_{x \in A} f(x)$ for all subsets A of Ω. Unfortunately, this construction does not work in the continuous case. Consider a simple *mixture of Gaussians*, here written in Fun (Borgström et al. 2011), a probabilistic functional language embedded within F# (Syme et al. 2007).

```
let w = {mA = 0.0; mB = 4.0} in
  if flip 0.7 then random(Gaussian(w.mA, 1.0)) else random(Gaussian(w.mB, 1.0))
```

This specifies a distribution on the real line (*i.e.* $\Omega = \mathbb{R}$) and corresponds to a generative process where one draws a number from a Gaussian distribution with precision 1.0, and with mean either 0.0 or 4.0 depending on the result of flipping a biased coin. We use a record w with fields mA and mB to hold each mean. Repeating the construction from the discrete case yields the function $g(x) = \mathbb{P}(\{x\})$, which is zero everywhere. Instead we look for a function f such that $\mathbb{P}(A) = \int_A f(x)\, dx$, known as the *probability density function* (PDF) of the distribution. In other words, f is a function where the area under its curve on an interval gives the probability of generating an outcome falling in that interval. The PDF of this program is given by the function

$$f(x) = 0.7 \cdot \mathsf{pdf\_Gaussian}(0.0,\, 1.0,\, \mathsf{x}) + 0.3 \cdot \mathsf{pdf\_Gaussian}(4.0,\, 1.0,\, \mathsf{x})$$

where pdf_Gaussian is the PDF of the Gaussian distribution, the famous "bell curve" from statistics. The function, pictured below, takes higher values where the generative process described above is more likely to generate an outcome.



Densities Functions and **MCMC.** In the example above, the means and variances of the Gaussians, as well as the bias between the two, were known. In this case, the PDF gives a measure of how likely a particular output is. The more common and interesting case in applications is where the parameters are *unknown*, but we have a sample from the process in question. In that case, evaluating the PDF at the sample gives the likelihood of the parameters: a measure of how well a given setting of the parameters matches the sample. We are often interested in properties of the function that maps parameters to their likelihood, *e.g.*, its maximum.

In Bayesian modelling, we use a prior distribution representing our prior beliefs on what the parameters are. Incidentally, this distribution also involves Gaussians, but with a low precision (high variance). To illustrate this, we modify our example as follows:

```
let prior () =
  { mA = random(Gaussian(0.0, 0.001)); mB = random(Gaussian(0.0, 0.001)) }
let moG w =
  if flip 0.7 then random(Gaussian(w.mA, 1.0)) else random(Gaussian(w.mB, 1.0))
let gen w = [| for i in 1 .. 1000 → moG w |]
```

This model generates an array of independent, identically distributed (i.i.d.) points, defined in terms of the single-point model. This lets us capture the idea of seeing many samples generated from the same process.

Markov chain Monte Carlo (MCMC) methods, which generate samples from the posterior distribution, are commonly used for probabilistic inference. The idea of MCMC is to construct a Markov chain in the parameter space of the model, whose equilibrium distribution is the posterior distribution over model parameters. Neal (1993) gives an excellent review of MCMC methods. We here use Filzbach (Purves and Lyutsarev 2012), an adaptive MCMC sampler based on the Metropolis-Hastings algorithm. All that is required for such algorithms is the ability to calculate the posterior density given a set of parameters. The posterior does not need to be from a mathematically convenient family of distributions. Samples from the posterior can then serve as its representation, or be used to calculate marginal distributions of parameters or other integrals under the posterior distribution.

The posterior density is a function of the PDFs of the various pieces of the model, so to perform inference using MCMC, we also need functions to compute the PDFs:

```
let pdf_prior () w = pdf_Gaussian(0.0, 0.001, w.mA) ∗ pdf_Gaussian(0.0, 0.001, w.mB)
let pdf_moG w x = 0.7 ∗ pdf_Gaussian(w.mA, 1.0, x) + 0.3 ∗ pdf_Gaussian(w.mB, 1.0, x)
let pdf_gen w xs = product [| for x in xs → pdf_moG w x |]
```

Filzbach and other MCMC libraries require users to write these three functions, in addition to the probabilistic generative functions prior and gen, which are used for model validation. The goal of this paper is to instead compile these density functions from the generative code. This relieves domain experts from having to write the density code in the first place, as well as from the error-prone task of manually keeping their model code and their density code in synch. Instead, both the PDF and synthetic data are derived from the same declarative specification of the model.

Contributions of this Paper. This work defines and applies automated techniques for computing densities to actual inference problems from various scientific applications. The primary technical contribution is a *density compiler* that is correct, useful, and relatively simple and efficient. Specifically:

– We provide the first implementation of a density compiler based on the specification by Bhat et al. (2012). We compile programs in the probabilistic language Infer.NET Fun (described in Section 2) to their corresponding density functions (Section 3).
– We prove that the compilation algorithm is sound (Theorem 1). This is the first such proof for any variant of this compiler.
– We show that the compiler greatly reduces the development effort of domain experts by freeing them from writing densities and that the produced code is comparable in performance to functions hand-coded by experts. We show this on textbook examples and on problems from ecology (Section 4).

2 Fun: Probabilistic Expressions (Review)

We use a version of the core calculus Fun (Borgström et al. 2011) with discrete observations only (implemented using a **fail** construct (Kiselyov and Shan 2009)). Fun is a first-order functional language without recursion that extends the language of Ramsey and Pfeffer (2002), and has a natural semantics in the sub-probability monad. Our implementation efficiently supports a richer language with arrays and array comprehensions, which can be given a semantics in this core.

We use base types **int**, **double** and **unit**, product types (denoting pairs) and sum types (denoting disjoint unions). We let c range over constant data of base type, n over integers and r over real numbers. We write $\mathrm{ty}(c) = t$ to mean that constant c has type t.

Types of Fun:

$t, u ::= \mathbf{int} \mid \mathbf{double} \mid \mathbf{unit} \mid (t_1 * t_2) \mid (t_1 + t_2)$

We take $\mathbf{bool} \triangleq \mathbf{unit} + \mathbf{unit}$. We assume a collection of total deterministic functions on these types, including arithmetic and logical operators. For totality, we devine divison by zero to yield zero, i.e. $r/0.0 \triangleq 0.0$. Each operation f of arity n has a signature of the form **val** $f\colon t_1 * \cdots * t_n \to t_{n+1}$. We also assume standard families of primitive probability distributions of type $\mathsf{PDist}\langle t\rangle$, including the following.

Distributions: Dist : $(x_1 : t_1 * \cdots * x_n : t_n) \to \mathsf{PDist}\langle t\rangle$

$\mathsf{Bernoulli} : (\mathsf{bias} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{bool}\rangle$
$\mathsf{Poisson} : (\mathsf{rate} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{int}\rangle$
$\mathsf{Gaussian} : (\mathsf{mean} : \mathbf{double} * \mathsf{prec} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{double}\rangle$
$\mathsf{Beta} : (\mathsf{a} : \mathbf{double} * \mathsf{b} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{double}\rangle$
$\mathsf{Gamma} : (\mathsf{shape} : \mathbf{double} * \mathsf{scale} : \mathbf{double}) \to \mathsf{PDist}\langle\mathbf{double}\rangle$

A Bernoulli distribution corresponds to a biased coin flip. The Poisson distribution describes the number of occurrences of independent events that occur at a given average rate. We parameterize the Gaussian distribution by mean and precision. The *standard deviation* σ follows from the identity $\sigma^2 = 1/\mathsf{prec}$. The Beta distribution is a suitable prior for the parameter of Bernoulli distributions; similarly the Gamma distribution is a suitable prior for the parameter of Poisson and the prec parameter of Gaussian.

Expressions of Fun:

$V ::= x \mid c \mid (V,V) \mid \mathbf{inl}_u V \mid \mathbf{inr}_t V$	value
$M, N ::=$	expression
$x \mid c \mid \mathbf{inl}_u\ M \mid \mathbf{inr}_t\ M \mid (M,N)$	value constructors
$\mathbf{fst}\ M \mid \mathbf{snd}\ M$	left/right projection from pair
$f(M)$	primitive operation (deterministic)
$\mathbf{let}\ x = M\ \mathbf{in}\ N$	let (scope of x is N)
$\mathbf{match}\ M\ \mathbf{with}\ \ \mathbf{inl}\ x_1 \to N_1 \mid \mathbf{inr}\ x_2 \to N_2$	matching (scope of x_i is N_i)
$\mathbf{random}(\mathrm{Dist}(M))$	primitive distribution
$\mathbf{fail}_t$	failure

To ensure that a program has at most one type in a given typing environment, **inl** and **inr** are annotated with a type (see (FUN INL) below). The expression **fail** is annotated with the type it is used at. We omit these types where they are not used. When X is a term (possibly with binders), we write $x_1,\ldots,x_n \sharp X$ if none of the x_i appear free in X. We let op(M) range over $f(M)$, **fst** M, **snd** M, **inl** M and **inr** M; () is the **unit** constant.

We write **observe** M for **if** M **then** () **else fail** and Uniform for Beta(1.0,1.0). When M has sum type, we write **if** M **then** N_1 **else** N_2 for **match** M **with inl** $\_ \to N_1 \mid$ **inr** $\_ \to N_2$.

We write $\Gamma \vdash M : t$ to mean that in type environment $\Gamma = x_1 : t_1, \ldots, x_n : t_n$ (x_i distinct) expression M has type t. Apart from the following, the typing rules are standard. In (FUN INL), (FUN INR) (not shown) and (FUN FAIL), type annotations are used in order to obtain a unique type. In (FUN RANDOM), a random variable drawn from a distribution of type $(x_1 : t_1 * \cdots * x_n : t_n) \to \mathsf{PDist}\langle t\rangle$ has type t.

Selected Typing Rules: $\Gamma \vdash M : t$

(FUN INL)
$$\frac{\Gamma \vdash M : t}{\Gamma \vdash \mathbf{inl}_u\ M : t + u}$$

(FUN FAIL)
$$\frac{}{\Gamma \vdash \mathbf{fail}_t : t}$$

(FUN RANDOM)
Dist : $(x_1 : t_1 * \cdots * x_n : t_n) \to \mathsf{PDist}\langle t\rangle$
$$\frac{\Gamma \vdash M : (t_1 * \cdots * t_n)}{\Gamma \vdash \mathbf{random}(\mathrm{Dist}(M)) : t}$$

Semantics. As usual, for precision concerning probabilities over uncountable sets, we turn to measure theory. The interpretation of a type t is the set $\mathbf{V}_t$ of closed values of type t (real numbers, integers etc.). Below we consider only Lebesgue-measurable sets of values, defined using the standard (Euclidian) metric, and ranged over by A, B.

A measure μ over t is a function, from (measurable) subsets of $\mathbf{V}_t$ to the non-negative real numbers extended with ∞, that is σ-additive, that is, $\mu(\varnothing) = 0.0$ and $\mu(\cup_i A_i) = \Sigma_i \mu(A_i)$ if $A_1, A_2, \ldots$ are pair-wise disjoint. The measure μ is called a probability measure if $\mu(\mathbf{V}_t) = 1.0$, and a sub-probability measure if $\mu(\mathbf{V}_t) \leq 1.0$.

We associate a default or *stock* measure to each type, inductively defined as the counting measure on $\mathbb{Z}$ and $\{()\}$, the Lebesgue measure on $\mathbb{R}$, and the Lebesgue-completion of the product and disjoint sum, respectively, of the two measures for $t * u$ and $t + u$. If f is a non-negative (measurable) function $t \to$ **double**, we let $\int f$ be the Lebesgue integral of f with respect to the stock measure on t, if the integral is defined. This integral coincides with $\Sigma_{x\in\mathbf{V}_t} f(x)$ for discrete types t, and with the standard Riemann integral (if it is defined) on $t =$ **double**. We write $\int f(x)\,dx$ for $\int \lambda x.f(x)$, and $\int f(x)\,d\mu(x)$ for Lebesgue integration with respect to the measure μ on t. The Iverson brackets $[p]$ are 1.0 if predicate p is true, and 0.0 otherwise. We write $\int_A f$ for $\int \lambda x.[x \in A] \cdot f(x)$. Let g be a *density* of μ (with respect to the stock measure) if $\int_A 1\,d\mu(x) = \int_A g$ for all A. If μ is a (sub-)probability measure, then we say that g as above is its PDF.

The semantics of a closed Fun expression M is a sub-probability measure $\mathbb{P}_M$ over its return type. Open **fail**-free Fun expressions have a straightforward semantics (Ramsey and Pfeffer 2002) in the probability monad (Giry 1982). In order to treat the **fail** primitive, our extension (Gordon et al. 2013) of Ramsey and Pfeffer's semantics uses a richer monad: the sub-probability monad (Panangaden 1999)[1]. In the sub-probability monad, bind and return are defined in the same way as the probability monad; it is only the set of admissible measures μ that is extended to admit $|\mu| \leq 1$. The semantics of an expression M is a sub-probability measure. Below, σ is a substitution, that gives values to the free variables of M.

Monadic Semantics of Fun with fail, $\mathscr{P}[\![M]\!]\ \sigma$: assuming $z \sharp N, N_1, x, x_1, x_2, \sigma$

$(\mu \gg= f)\ A \triangleq \int f(x)(A)\, d\mu(x)$	Monadic bind
$(\texttt{return}\, V)\ A \triangleq 1$ if $V \in A$, else 0	Monadic return
$\texttt{zero}\, A \triangleq 0$	Monadic zero

$$\mathscr{P}[\![x]\!]\ \sigma \triangleq \texttt{return}\ (x\sigma)$$
$$\mathscr{P}[\![c]\!]\ \sigma \triangleq \texttt{return}\, c$$
$$\mathscr{P}[\![\mathrm{op}(M)]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg= \texttt{return} \circ \mathrm{op}$$
$$\mathscr{P}[\![(M,N)]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg= \lambda z.\mathscr{P}[\![N]\!]\ \sigma \gg= \lambda w.\texttt{return}\ (z,w)$$
$$\mathscr{P}[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg= \lambda z.\mathscr{P}[\![N]\!]\ (\sigma, x \mapsto z)$$
$$\mathscr{P}[\![\mathbf{match}\ M\ \mathbf{with}\ \mathbf{inl}\ x_1 \to N_1 \mid \mathbf{inr}\ x_2 \to N_2]\!]\ \sigma \triangleq$$
$$\quad \mathscr{P}[\![M]\!]\ \sigma \gg= \texttt{either}\ (\lambda z.\mathscr{P}[\![N_1]\!]\ (\sigma, x_1 \mapsto z))\ (\lambda z.\mathscr{P}[\![N_2]\!]\ (\sigma, x_2 \mapsto z))$$
$$\mathscr{P}[\![\mathbf{random}(\mathrm{Dist}(M))]\!]\ \sigma \triangleq \mathscr{P}[\![M]\!]\ \sigma \gg= \lambda z.\mu_{\mathrm{Dist}(z)}$$
$$\mathscr{P}[\![\mathbf{fail}]\!]\ \sigma \triangleq \texttt{zero}$$

Here $\texttt{either}\ f\ g\ (\mathbf{inl}\ V) \triangleq f\ V$ and $\texttt{either}\ f\ g\ (\mathbf{inr}\ V) \triangleq g\ V$. We let the semantics of a closed expression M be $\mathbb{P}_M \triangleq \mathscr{P}[\![M]\!]\ \varepsilon$, where ε denotes the empty substitution.

3 The Density Compiler

We compute the PDF of a Fun program by compilation into a deterministic language, that features integration as a primitive operation. In our implementation, we call out to a numeric integration library to compute the value of integrals. Our compilation is based on that of Bhat et al. (2012), with modifications to treat **fail** statements, deterministic let bindings, **match** (and general **if**) statements, and integer arithmetic.

3.1 Target Language for Density Computations

For our target language, we choose a standard deterministic functional language, augmented with stock integration.

Expressions of the Target Language: E, F

$T, U ::= \mathbf{int} \mid \mathbf{double} \mid \mathbf{unit} \mid T \to U \mid T + U \mid T * U$	target types
$E, F ::=$	target expression
$\quad x \mid c \mid \mathbf{inl}_U\ E \mid \mathbf{inr}_T\ E \mid (E, F)$	value constructors

[1] Sub-probabilities are also useful to reason about our compilation of **match** (and **if**) statements, where the probability that we have entered a particular branch may be less than 1.

$\mathbf{fst}\ E \mid \mathbf{snd}\ E \mid f(E)$	deterministic operations
$\mathbf{let}\ x = E\ \mathbf{in}\ F$	let (scope of x is F)
$\mathbf{match}\ E\ \mathbf{with}\ \ \mathbf{inl}\ x_1 \to F_1 \mid \mathbf{inr}\ x_2 \to F_2$	matching (scope of x_i is F_i)
$\lambda(x_1,...,x_n).\ E$	lambda abstraction
$E\ F$	application
$\int E$	stock integration
$\bot_T$	failure

The typing rules for integration and failure are as follows (the other typing rules are standard):

Selected Typing Rules: $\Gamma \vdash E : T$

(TARGET INT)
$$\frac{\Gamma \vdash E : T \to \mathbf{double} \qquad T \text{ is a first-order type}}{\Gamma \vdash \int E : \mathbf{double}}$$

(TARGET FAIL)
$$\frac{}{\Gamma \vdash \bot_T : T}$$

Small-step CBV-evaluation $\to$ of well-typed expressions is standard, except for short-circuiting multiplication: $0.0 \cdot E \to 0.0$, avoiding failures in E. Evaluation can fail either explicitly ($\bot$) or by evaluating an undefined integral, *e.g.* $\int \lambda x.\sin x \to \bot_{\mathbf{double}}$.

3.2 Relational Specification of the Compiler

The translation is based on the let-structure of the expression. Variables that are let-bound in outer lets are referred to as parameters, and a context gathers random and deterministic inner lets.

Probability Context:

$\Upsilon ::=$	probability context
ε	empty context
Υ, x	random variable
$\Upsilon, x = E$	deterministic variable

A probabilistic context Υ is often used together with a density expression (E below), which is an open term that expresses the joint probability density of the random variables in the context and the constraints that have been collected when choosing branches in **match** statements. The main judgment is $\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F$, which computes a function F from return values of M to densities, where parameters may occur free in F. The marginal judgment $\Upsilon; E \vdash \mathrm{marg}(x_1,\ldots,x_k) \Rightarrow F$ yields the joint PDF of its argument, marginalizing out all other random variables in Υ.

Inductively Defined Judgments of the Compiler:

$\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F$	in $\Upsilon; E$ expression F gives the PDF of M
$\Upsilon; E \vdash \mathrm{marg}(x_1,\ldots,x_k) \Rightarrow F$	in $\Upsilon; E$ expression F gives the PDF of $(x_1,\ldots,x_k)$

For a probability context to be well-formed, it has to be well-scoped and well-typed.

Well-formed probability context: $\Gamma \vdash \Upsilon \text{ wf}$

(ENV EMPTY)
$$\frac{}{\Gamma \vdash \varepsilon \text{ wf}}$$

(ENV VAR)
$$\frac{\Gamma \vdash \Upsilon \text{ wf} \qquad \Gamma \vdash x : t \qquad x \sharp \Upsilon}{\Gamma \vdash \Upsilon, x \text{ wf}}$$

(ENV CONST)
$$\frac{\Gamma \vdash \Upsilon \text{ wf} \qquad \Gamma \vdash x : t \qquad x \sharp \Upsilon \qquad \Gamma \vdash E : t}{\Gamma \vdash \Upsilon, x = E \text{ wf}}$$

Given a well-formed context Υ, we can extract the random variables $\text{rands}(\Upsilon)$, and an idempotent substitution σ_Υ that describes the deterministic variables.

Random variables $\text{rands}(\Upsilon)$ **and values of deterministic variables** σ_Υ

$$\begin{aligned}
\text{rands}(\varepsilon) &\triangleq \varepsilon & \sigma_\varepsilon &\triangleq [] \\
\text{rands}(\Upsilon, x) &\triangleq \text{rands}(\Upsilon), x & \sigma_{\Upsilon,x} &\triangleq \sigma_\Upsilon \\
\text{rands}(\Upsilon, x = E) &\triangleq \text{rands}(\Upsilon) & \sigma_{\Upsilon,x=E} &\triangleq [x \mapsto E\sigma_\Upsilon]\sigma_\Upsilon
\end{aligned}$$

We define "M det" to hold iff M does not contain any occurrence of **random** or **fail**. If M det holds, then M is also an expression in the target language syntax, and we silently treat it as such (in rules (LET DET) and (MATCH DET), for example). If M det and $\text{rands}(\Upsilon) \sharp (M\sigma_\Upsilon)$, then M is constant under Υ.

The marg judgment yields the joint marginal PDF of the random variables in its argument. To compute the PDF, we first substitute in the deterministic **let**-bound variables, and then integrate out the remaining random variables. Except for rule (DISCRETE) below, $\text{marg}(x_1, \ldots, x_k)$ is used with $k \in \{0, 1, 2\}$; the case $k = 0$ is used to compute the probability of being in the current branch of the program.

Marginal Density: $\Upsilon; E \vdash \text{marg}(x_1, \ldots, x_k) \Rightarrow F$

(MARGINAL)
$$\frac{\{x_1, \ldots, x_k\} \cup \{y_1, \ldots, y_n\} = \text{rands}(\Upsilon) \qquad x_1, \ldots, x_k, y_1, \ldots, y_n \text{ distinct}}{\Upsilon; E \vdash \text{marg}(x_1, \ldots, x_k) \Rightarrow \lambda(x_1, \ldots, x_k).\ \int \lambda(y_1, \ldots, y_n).\ E\sigma_\Upsilon}$$

The dens judgment gives the density F of M in the current context Υ, where E is the accumulated body of the density function so far. We introduce fresh lambda-bound variables in the result F; below we assume that $z, w \sharp \Upsilon, E, M$.

Density Compiler, base cases: $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$

(VAR DET)
$$\frac{(x = E') \in \Upsilon \qquad \Upsilon; E \vdash \text{dens}(E') \Rightarrow F}{\Upsilon; E \vdash \text{dens}(x) \Rightarrow F}$$

(VAR RND)
$$\frac{x \in \text{rands}(\Upsilon) \qquad \Upsilon; E \vdash \text{marg}(x) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(x) \Rightarrow F}$$

(CONSTANT)
$$\frac{\text{ty}(c) \text{ countable} \qquad \Upsilon; E \vdash \text{marg}(\varepsilon) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(c) \Rightarrow \lambda z.\ [z = c] \cdot (F\ ())}$$

(FAIL)
$$\frac{}{\Upsilon; E \vdash \text{dens}(\textbf{fail}) \Rightarrow \lambda z.\ 0.0}$$

For a deterministic variable, (VAR DET) recurses into its definition. The rule (VAR RND) computes the marginal density of a random variable using the marg judgment. The (CONSTANT) rule states that the probability density of a discrete constant c (built from sums and products of integers and units) is the probability of being in the current branch at c, and 0 elsewhere. The (FAIL) rule gives that the density of **fail** is zero.

Density Compiler, sums and tuples: $\Upsilon;E \vdash \mathrm{dens}(M) \Rightarrow F$

(SUM CON L)
$$\frac{\Upsilon;E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon;E \vdash \mathrm{dens}(\mathbf{inl}\ M) \Rightarrow \mathsf{either}\ F\ (\lambda\_.0)}$$

(FROML)
$$\frac{\Upsilon;E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon;E \vdash \mathrm{dens}(\mathsf{fromL}(M)) \Rightarrow \lambda z.(F\ (\mathbf{inl}\ z))}$$

(TUPLE VAR)
$$\frac{\Upsilon;E \vdash \mathrm{marg}(x,y) \Rightarrow F}{\Upsilon;E \vdash \mathrm{dens}((x,y)) \Rightarrow F}$$

(TUPLE PROJ L)
$$\frac{\Upsilon;E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon;E \vdash \mathrm{dens}(\mathbf{fst}\ M) \Rightarrow \lambda z.\ \textstyle\int \lambda w.\ F\ (z,w)}$$

Symmetric versions of (SUM CON L), (TUPLE PROJ L) and (FROML) are omitted above. (SUM CON L) states that the density of **inl** M is the density of M in the left branch of a sum, and 0 in the right. Its dual is (FROML). The rule (TUPLE VAR) computes the joint marginal density of two random variables. (This syntactic restriction can be lifted by considering dependency information for the expressions in the tuple (Bhat et al. 2012).) (TUPLE PROJ L) marginalizes out the left dimension of a pair.

Density Compiler, let and match: $\Upsilon;E \vdash \mathrm{dens}(\mathbf{let}\ x = M\ \mathbf{in}\ N) \Rightarrow F$

(LET DET)
$$\frac{M\ \det \qquad \Upsilon, x = M;E \vdash \mathrm{dens}(N) \Rightarrow F}{\Upsilon;E \vdash \mathrm{dens}(\mathbf{let}\ x = M\ \mathbf{in}\ N) \Rightarrow F}$$

(LET RND)
$$\frac{\neg(M\ \det) \qquad \varepsilon;1 \vdash \mathrm{dens}(M) \Rightarrow F_1 \qquad \Upsilon, x;E \cdot (F_1\ x) \vdash \mathrm{dens}(N) \Rightarrow F_2}{\Upsilon;E \vdash \mathrm{dens}(\mathbf{let}\ x = M\ \mathbf{in}\ N) \Rightarrow F_2}$$

The rule (LET DET) simply adds a deterministic let-binding to the context. In (LET RND), we compute the density of the let-bound variable in an empty context, and multiply it into the current accumulated density when computing the density of the body.

Below, we let $\mathsf{isL} := \lambda x.\mathbf{if}\ x\ \mathbf{then}\ 1.0\ \mathbf{else}\ 0.0$ be the indicator function for the left branch, and dually for isR. We also use a deterministic operation $\mathsf{fromL} : t + u \to t$ such that $\mathsf{fromL}(M) \to \mathbf{match}\ M\ \mathbf{with}\ \mathbf{inl}\ x \to\ x \mid \mathbf{inr}\ y \to\ \bot_t$, and its dual fromR.

Density Compiler, rules for match: $\Upsilon;E \vdash \mathrm{dens}(\mathbf{match}\ M\ \mathbf{with} \ldots) \Rightarrow F$

(MATCH DET)
$$\frac{M\ \det \qquad \begin{array}{l}\Upsilon, y_1 = \mathsf{fromL}(M);E \cdot (\mathsf{isL}\ M\sigma_\Upsilon) \vdash \mathrm{dens}(N_1) \Rightarrow F_1 \\ \Upsilon, y_2 = \mathsf{fromR}(M);E \cdot (\mathsf{isR}\ M\sigma_\Upsilon) \vdash \mathrm{dens}(N_2) \Rightarrow F_2\end{array}}{\Upsilon;E \vdash \mathrm{dens}(\mathbf{match}\ M\ \mathbf{with}\ \mathbf{inl}\ y_1 \to\ N_1 \mid \mathbf{inr}\ y_2 \to\ N_2) \Rightarrow \lambda z.\ (F_1\ z) + (F_2\ z)}$$

(MATCH RND)
$$\frac{\begin{array}{c}\neg(M\ \det) \\ \varepsilon;1 \vdash \mathrm{dens}(M) \Rightarrow F\end{array} \qquad \begin{array}{l}\Upsilon, y_1;E \cdot (F\ (\mathbf{inl}\ y_1)) \vdash \mathrm{dens}(N_1) \Rightarrow F_1 \\ \Upsilon, y_2;E \cdot (F\ (\mathbf{inr}\ y_2)) \vdash \mathrm{dens}(N_2) \Rightarrow F_2\end{array}}{\Upsilon;E \vdash \mathrm{dens}(\mathbf{match}\ M\ \mathbf{with}\ \mathbf{inl}\ y_1 \to\ N_1 \mid \mathbf{inr}\ y_2 \to\ N_2) \Rightarrow \lambda z.\ (F_1\ z) + (F_2\ z)}$$

(MATCH DET) is based on (LET DET), and we multiply the constraint that we are in the correct branch ($\mathsf{isL}\ M\sigma_\Upsilon$ or $\mathsf{isR}\ M\sigma_\Upsilon$) with the joint density expression. We also employ deterministic functions fromL and fromR to avoid recursive calls to (MATCH DET) when computing the density of the match-bound variable. The (MATCH RND) rule is based on (LET RND), and we again multiply in the constraint that we are in the left (or right) branch of the **match**.

Density Compiler, random variables : $\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F$

(RANDOM CONST)

$$\frac{M\ \mathrm{det} \quad \mathrm{rands}(\Upsilon) \sharp (M\sigma_\Upsilon) \quad \Upsilon; E \vdash \mathrm{marg}(\varepsilon) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(\mathbf{random}(\mathrm{Dist}(M))) \Rightarrow \lambda z.\ (\mathrm{pdf}_{\mathrm{Dist}(M\sigma_\Upsilon)}\ z) \cdot (F\ ())}$$

(RANDOM RND)

$$\frac{\neg(M\ \mathrm{det} \wedge \mathrm{rands}(\Upsilon) \sharp (M\sigma_\Upsilon)) \quad \Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(\mathbf{random}(\mathrm{Dist}(M))) \Rightarrow \lambda z. \textstyle\int \lambda w.(\mathrm{pdf}_{\mathrm{Dist}(w)}\ z) \cdot (F\ w)}$$

In (RANDOM CONST), a random variable drawn from a primitive distribution with a constant argument has the expected PDF (multiplied with the probability that we are in the current branch). (RANDOM RND) treats calls to **random** with a random argument by marginalizing over the argument to the distribution.

In **if** statements, the branching expression is of type **bool** $=$ **unit** $+$ **unit**, so we can make a straightforward case distinction.

Derived rule for if statements

(IF DET)

$$\frac{M\ \mathrm{det} \quad \Upsilon; E \cdot [M\sigma_\Upsilon = \mathbf{true}] \vdash \mathrm{dens}(N_1) \Rightarrow F_1 \quad \Upsilon; E \cdot [M\sigma_\Upsilon = \mathbf{false}] \vdash \mathrm{dens}(N_2) \Rightarrow F_2}{\Upsilon; E \vdash \mathrm{dens}(\mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2) \Rightarrow \lambda z.\ (F_1\ z) + (F_2\ z)}$$

For numeric operations on real numbers we mimic the change of variable rule of integration (often summarized as "$dx = \frac{dx}{dy} dy$"), multiplying the density of the argument with the derivative of the inverse operation. This is exemplified by the following rules.

Density compiler, numeric operations on reals : $\Upsilon; E \vdash \mathrm{dens}(f(M)) \Rightarrow F$

(NEG)

$$\frac{\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(-M) \Rightarrow \lambda z.\ F\ (-z)}$$

(INVERSE)

$$\frac{\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(1/M) \Rightarrow \lambda z.\ (F\ 1/z) \cdot (1/z^2)}$$

(EXP)

$$\frac{\Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(\mathsf{exp}(M)) \Rightarrow \lambda z.\ \mathbf{if}\ z > 0.0\ \mathbf{then} (F\ \mathsf{log}(z)) \cdot (1/z)\ \mathbf{else}\ 0.0}$$

(TRANSLATE)

$$\frac{N\ \mathrm{det} \quad \mathrm{rands}(\Upsilon) \sharp (N\sigma_\Upsilon) \quad \Upsilon; E \vdash \mathrm{dens}(M) \Rightarrow F}{\Upsilon; E \vdash \mathrm{dens}(M + N) \Rightarrow \lambda z.\ F\ (z - N\sigma_\Upsilon)}$$

(PLUS)

$$\frac{\Upsilon; E \vdash \text{dens}((M,N)) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(M+N) \Rightarrow \lambda z.\ \int \lambda w.\ F\ (w, z-w)}$$

The (DISCRETE) rule for discrete operations such as logical and comparison operations and integer arithmetic computes the expectation of an indicator function over the joint distribution of the random variables occurring in the expression.

Density compiler, discrete operations : $\Upsilon; E \vdash \text{dens}(f(M)) \Rightarrow F$

(DISCRETE)

$$\frac{f : t \to u \quad u \text{ discrete} \quad M \text{ det} \quad \bar{y} = \text{rands}(\Upsilon) \cap \text{fv}(M\sigma_\Upsilon) \quad \Upsilon; E \vdash \text{marg}(\bar{y}) \Rightarrow F}{\Upsilon; E \vdash \text{dens}(f(M)) \Rightarrow \lambda z.\ \int \lambda \bar{y}.\ [z = f(M\sigma_\Upsilon)] \cdot (F\ \bar{y})}$$

These derived judgments relate the types of the various terms occurring in the marg and dens judgments.

Lemma 1 (Derived Judgments)
If $\Gamma, \Gamma_\Upsilon \vdash \Upsilon$ wf *and* $\text{dom}(\Gamma_\Upsilon) = \text{rands}(\Upsilon) \cup \text{dom}(\sigma_\Upsilon)$ *and* $\Gamma, \Gamma_\Upsilon \vdash E : \textbf{double}$ *then*

(1) *If* $\Upsilon; E \vdash \text{marg}(x_1, \dots, x_n) \Rightarrow F$ *and* $\Gamma_\Upsilon \vdash (x_1, \dots, x_n) : (t_1 * \cdots * t_n)$ *then* $\Gamma \vdash F : (t_1 * \cdots * t_n) \to \textbf{double}$.
(2) *If* $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$ *and* $\Gamma, \Gamma_\Upsilon \vdash M : t$ *then* $\Gamma \vdash F : t \to \textbf{double}$.

The soundness theorem asserts that, for all closed expressions M, the density function given by the density compiler indeed characterizes (via stock integration) the distribution of M given by the monadic semantics:

Theorem 1 (Soundness). *If* $\varepsilon; 1 \vdash \text{dens}(M) \Rightarrow F$ *and* $\varepsilon \vdash M : t$ *then*

$$(\mathscr{P}[\![M]\!]\ \varepsilon)\ A = \int_A F$$

Proof: By joint induction on the derivations of $\text{dens}(M)$ and $M : t$, using the following induction hypothesis: if $\Gamma, \Gamma_\Upsilon \vdash \Upsilon$ wf and $\Upsilon; E \vdash \text{dens}(M) \Rightarrow F$ and $\Gamma, \Gamma_\Upsilon \vdash M : t$ and $\Gamma, \Gamma_\Upsilon \vdash E : \textbf{double}$ and $\Gamma \vdash \rho$ and $\text{dom}(\Gamma_\Upsilon) = \text{rands}(\Upsilon) \cup \text{dom}(\sigma_\Upsilon)$ and $|\mu| \leq 1$ and $\mu(B) = \int_B \lambda(\text{rands}(\Upsilon)).E\rho$, and $(\forall x \in \text{dom}(\sigma_\Upsilon) \forall \rho'.\ \Gamma_\Upsilon \vdash \rho'$ and $\sigma_\Upsilon(x)\rho\rho' \to^* \bot$ implies that $E\rho\rho' \to^* 0.0)$ then

$$(\mu \gg\!= (\lambda(\text{rands}(\Upsilon)).(\mathscr{P}[\![M]\!]\ (\sigma_\Upsilon \rho))))\ A = \int_A F\rho$$

where $\Gamma \vdash \rho$ is defined as $\varepsilon \vdash \varepsilon$, and $\Gamma, x : t \vdash \rho[x \mapsto V]$ when $\varepsilon \vdash V : t$ and $\Gamma \vdash \rho$. ■

The induction hypothesis on evaluation of $\sigma_\Upsilon(x)\rho\rho'$ above is used when attempting to evaluate match-bound variables for valuations that give the other branch. For such valuations the density becomes zero, because of the short-circuiting property of multiplication by 0.0.

As an example of compilation, the **if** statement in the program

let p = **random**(Uniform) **in let** b = **random**(Bernoulli(p)) **in if** b **then** p+1.0 **else** p

is handled by (IF DET), yielding a density function that is β-equivalent to

$$\begin{aligned}\lambda z. \quad & \int \lambda b.[0 \leq z-1 \leq 1] \cdot (\textbf{if } b \textbf{ then } z-1 \textbf{ else } 2-z) \cdot [b = \textbf{true}] \\ + & \int \lambda b.[0 \leq z \leq 1] \cdot (\textbf{if } b \textbf{ then } z \textbf{ else } 1-z) \cdot [b = \textbf{false}]\end{aligned}$$

which simplifies to the (V-shaped) function $\lambda z.[1 \leq z \leq 2] \cdot (z-1) + [0 \leq z \leq 1] \cdot (1-z)$.

4 Evaluation

We evaluate the compiler on several synthetic textbook examples and several real examples from scientific applications. We wish to validate that the density compiler handles these examples, and understand how much the compiler reduces the developer burden, and its performance impact.

Implementation. Since Fun is a sublanguage of F#, we implement our models as F# programs, and use the quotation mechanism of F# to capture their syntax trees. Running the F# program corresponds to sampling data from the model. To compute the PDF, the compiler takes the syntax tree (of F# type Expr) of the model and produces another Expr corresponding to a deterministic F# program as output. We then use run-time code generation to compile the generated Expr to MSIL bytecode, which is just-in-time compiled to executable machine code when called, just as for statically compiled F# code. Our implementation supports arrays and records, which are both translated using adaptations of the corresponding rules for tuples. For efficiency, the implementation must avoid introducing redundant computations, translating the use of substitution in the formal rules to more efficient **let**-bindings that share the values of expressions that would otherwise be re-computed. As is common practice, our implemenation and Filzbach both work with the *logarithm* of the density, which avoids products of densities in favor of sums of log-densities where possible, to avoid numerical underflow.

Metrics. We consider scientific models with existing implementations for MCMC-based inference, written by domain experts. We are interested in how the modelling and inference experience would change, in terms of developer effort and performance impact, when adopting the Fun-based solution.

We assess the reduction in developer burden by measuring the code sizes (in lines-of-code (LOC)) of the original implementations of model and density code, and of the corresponding Fun model. For the synthetic examples, we have written both the model and the density code. The original implementations of the scientific models contain helper code such as I/O code for reading and writing data files in an application-specific format. Our LOC counts do not consider such helper code, but only count the code for generating synthetic data from the model, code for computing the logarithm of the posterior density of the model, and model-related code for setting up and interacting

Table 1. Lines-of-code and running time comparisons of synthetic and scientific models

Example	orig	LOC, orig	LOC, Fun		time (s), orig	time (s), Fun	
mixture of Gaussians	F#	32	20	0.63x	1.77	4.78	2.7x
linear regression	F#	27	18	0.67x	0.63	2.08	3.3x
species distribution	C#	173	37	0.21x	79	189	2.4x
net primary productivity	C#	82	39	0.48x	11	23	2.1x
global carbon cycle	C#	1532	402	0.26x	n/a	764	n/a

with Filzbach itself. We also compare the running times of the original implementations versus the Fun versions for MCMC-based inference using Filzbach, not including data manipulation before and after running inference.

4.1 Examples

Synthetic examples. Our synthetic examples are models for two classic problems in statistics and machine learning: the supervised learning task *linear regression*, and the unsupervised learning task *mixture of Gaussians*. The latter can be thought of as a probabilistic version of *k-means clustering*. In linear regression, inference is trying to determine the coefficients of the line. In mixture of Gaussians, inference is trying to determine the unknown mixing bias and the means and variances of the Gaussian components.

Species Distribution. The species distribution problem is to give the probability that certain species will be present at a given site, based on climate factors. It is a problem of long-standing interest in ecology and has taken on new relevance in light of the issue of climate change. The particular model that we consider is designed to mitigate *regression dilution* arising from uncertainty in the predictor variables, for example, measurement error in temperature data (McInerny and Purves 2011). Inference tries to determine various features of the species and the environment, such as the optimal temperature preferred by a species, or the true temperature at a site.

Global Carbon Cycle. The dynamics of the Earth's climate are intertwined with the terrestrial carbon cycle, and better carbon models (modelling how carbon in the air gets converted to biomass) enable better constrained projections about these systems. We consider a fully data-constrained terrestrial carbon model by Smith et al. (2012). It is a composition of various submodels for smaller processes such as *net primary productivity*, the fine root mortality rate or the fraction of trees that are evergreen versus deciduous. Inference tries to determine the different parameters of these submodels.

Discussion. Table 1 reports the metrics for each example. The LOC numbers show significant reduction in code size, with more significant savings as the size of the model grows. The larger models (where the Fun versions are $\approx 25\%$ of the size of the original) are more indicative of the savings in developer and maintenance effort, since smaller models have a larger fraction of boiler-plate code. We find the running times encouraging: we have made little attempt to optimize the generated code, and preliminary testing indicates that much of the performance slow-down is due to constant factors.

The global carbon cycle model is composed of submodels, each with their own dataset. Unfortunately, it is unclear from the original source code how this composition translates to a run of inference, making it difficult to know what constitutes a fair comparison. Thus, we do not report a running time for the full model. However, we can measure the running time of individual submodels, such as net primary productivity, where the data and control flow are simpler.

5 Related Work

The most closely related work to this paper is recent work by Bhat et al. (2012) who develop a theoretical framework for computing PDFs, but describe no implementation nor correctness proof. The density compiler of Section 3 has a simpler presentation, with two judgments compared to five, and has rules for deterministic **let**s and operations on integers. Our paper also uses a richer language (Fun), which adds **fail**, **match** and general **if** (and for performance reasons, deterministic **let**).

Gordon et al. (2013) describe a naive density calculation routine for Fun without random **let**s; this sublanguage does not cover many useful classes of models such as hierarchical and mixture models.

The BUGS system computes densities from declaratively specified models to perform Gibbs sampling (Gilks et al. 1994). However, the models are not compositional as in this work, and only the joint density over all variables is possible. The AutoBayes system also computes densities for deriving maximum likelihood and Bayesian estimators for a significant class of statistical models (Schumann et al. 2008). It is not formally specified and does not appear to be compositional. Neither system addresses the non-existence of PDFs, presumably restricting expressivity in order to avoid the issue.

Inference for the Church language also uses MCMC, but works with distributions over the runs of a program instead of over its return value (Wingate et al. 2011).

6 Conclusions and Future Work

We have described a compiler for automatically computing probability density functions for programs from a rich Bayesian probabilistic programming language, proven the algorithm correct, and shown its applicability to real-world scientific models.

The inclusion of **fail** in the language appears highly useful for scientific models, giving a simple facility to exclude branches that are scientifically impossible from consideration. However, more investigation is needed to settle this claim.

Techniques from automatic differentiation (Griewank and Walther 2008) may be useful to treat higher-dimensional primitive probability distributions.

A drawback of the compiler is that terms of composite type are required either to have a PDF or to be deterministic, ruling out terms such as (0.0, **random**(Uniform)). One possibility for future work would be to refine the types of expressions with determinacy information, and make use of this additional information in the compiler.

References

Bhat, S., Agarwal, A., Vuduc, R.W., Gray, A.G.: A type theory for probability density functions. In: Field, J., Hicks, M. (eds.) POPL, pp. 545–556. ACM (2012)

Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., Van Gael, J.: Measure Transformer Semantics for Bayesian Machine Learning. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 77–96. Springer, Heidelberg (2011), http://research.microsoft.com/fun

Gilks, W.R., Thomas, A., Spiegelhalter, D.J.: A language and program for complex Bayesian modelling. The Statistician 43, 169–178 (1994)

Giry, M.: A categorical approach to probability theory. In: Banaschewski, B. (ed.) Categorical Aspects of Topology and Analysis. Lecture Notes in Mathematics, vol. 915, pp. 68–85. Springer, Heidelberg (1982)

Gordon, A.D., Aizatulin, M., Borgström, J., Claret, G., Graepel, T., Nori, A., Rajamani, S., Russo, C.: A model-learner pattern for Bayesian reasoning. In: POPL (2013)

Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd edn. SIAM (2008)

Kiselyov, O., Shan, C.-C.: Embedded Probabilistic Programming. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 360–384. Springer, Heidelberg (2009)

McInerny, G., Purves, D.: Fine-scale environmental variation in species distribution modelling: regression dilution, latent variables and neighbourly advice. Methods in Ecology and Evolution 2(3), 248–257 (2011)

Neal, R.M.: Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto (September 1993)

Panangaden, P.: The category of Markov kernels. Electronic Notes in Theoretical Computer Science 22, 171–187 (1999)

Purves, D., Lyutsarev, V.: Filzbach User Guide (2012), http://research.microsoft.com/en-us/um/cambridge/groups/science/tools/filzbach/filzbach.htm

Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: POPL, pp. 154–165 (2002)

Schumann, J., Pressburger, T., Denney, E., Buntine, W., Fischer, B.: AutoBayes program synthesis system users manual. Technical Report NASA/TM–2008–215366, NASA Ames Research Center (2008)

Smith, M.J., Vanderwel, M.C., Lyutsarev, V., Emmott, S., Purves, D.W.: The climate dependence of the terrestrial carbon cycle; including parameter and structural uncertainties. Biogeosciences Discussions 9, 13439–13496 (2012)

Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress (2007)

Wingate, D., Stuhlmueller, A., Goodman, N.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Proceedings of the 14th Intl. Conf. on Artificial Intelligence and Statistics, p. 131 (2011)

Polyglot: Systematic Analysis for Multiple Statechart Formalisms*

Daniel Balasubramanian[1], Corina S. Păsăreanu[2], Gábor Karsai[1], and Michael R. Lowry[1]

[1] Vanderbilt University
[2] Carnegie Mellon Silicon Valley
[3] NASA Ames
{daniel,gabor}@isis.vanderbilt.edu,
{corina.s.pasareanu,michael.r.lowry}@nasa.gov

Abstract. Polyglot is a tool for the systematic analysis of systems integrated from components built using multiple Statechart formalisms. In Polyglot, Statechart models are translated into a common Java representation with pluggable semantics for different Statechart variants. Polyglot is tightly integrated with the Java Pathfinder verification tool-set, providing analysis and test-case generation capabilities. The tool has been applied in the context of safety-critical software systems whose interacting components were modeled using multiple Statechart formalisms.

Keywords: Statecharts, symbolic execution, model checking.

1 Introduction and Tool Overview

Polyglot is a unified environment in which multiple variants of Statecharts [1], a popular modeling formalism for the dynamics of reactive systems, can be executed and verified against properties. The work on Polyglot has been motivated by large programs such as human space exploration, that involve multiple systems that interact via safety-critical protocols. These systems have been designed using *different* Statechart formalisms to build models from which code is automatically generated. Determining the impact of using different formalisms on the reliability and safety of such model-based software has been a daunting task with little prior tool support available.

Polyglot performs the analysis of the different models (e.g. expressed in Matlab Stateflow or Rational Rhapsody) by translating them to a common intermediate representation, which is then translated into Java code that represents the "structure" of the model (see Figure 1). The semantics are provided as separate "pluggable" modules. Currently, Polyglot includes modules that implement the semantics of Matlab Stateflow, Rational Rhapsody, and UML Statemachines; the framework can be extended easily with other Statechart semantics. The Java

* The rights of this work are transferred to the extent transferable according to title 17 U.S.C. 105.

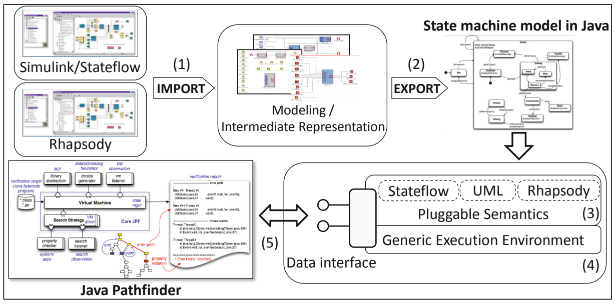


Fig. 1. The Polyglot tool

code representing the structure of the model is combined with one of these semantic modules, resulting in an executable component. We have also developed a formal description for the various Statecharts semantics using the structural operational semantics formalism (SOS) [2] to provide confidence in our implementation. Properties of interest are expressed using *specification patterns* [3] which are automatically translated into checking code similar to observer automata [4]. The analysis is performed using Java Pathfinder (JPF) [5]. JPF is a mature open-source tool-set for the verification of Java bytecode, that incorporates model checking and powerful test-case generation (i.e. the symbolic execution tool Symbolic PathFinder – SPF [6]) and compositional verification capabilities [7]. Polyglot is written in Java and it is freely available from [8].

The clear separation between the model structure and the different semantics provides several advantages. First, it provides the basis for analyzing interacting models that operate under different semantics. This is crucial to finding interoperability and interface errors early in the design phase, since e.g. previous findings show that the majority of errors in NASA's Apollo and Skylab software were interface errors [9]. Furthermore, this approach allows users to verify whether model properties are preserved across different variants of Statecharts, ensuring that there are no misunderstandings in requirements and design development due to semantical differences. Moreover, Polyglot allows a user to understand and analyze the behavior of models across different tools in a single framework.

Verification and validation techniques exist for several individual modeling formalisms, and supporting tools offer features such as test-input generation and model checking (see below). However, existing modeling languages and analysis tools are limited to a single Statechart formalism and have limited verification capabilities. What distinguishes Polyglot from other related approaches is its *extensibility* both in terms of Statechart semantics that are supported (via "pluggable" semantics) and analyses that can be performed, via the extensible JPF verification framework or custom analysis.

Related Tools. The analysis of Simulink/Stateflow models is supported by commercial tools such as Mathworks' *Design Verifier*, used for model checking and test case generation, and Reactive System's *Reactis* and T-VEC's *tester*, used for test generation and coverage. Similarly, for UML Statecharts, there are a wide variety of research tools. However, we believe that the ability to analyze multiple semantics in one environment is a major benefit to our approach.

Polyglot is similar to the heterogeneous model analysis from [10], which is based on a common "inframodel" and a set of rules describing the semantics and interactions between multiple formalisms. The work is concerned with high-level model descriptions and it would take considerable effort to use those rules to capture the semantic details for the Statecharts that are the focus here. Also that work does not address property preservation under different semantics.

The Ptolemy environment [11] is a laboratory for experimenting with different models of computation for component based systems. Ptolemy implements polymorphic components whose behavioral semantics depend on an "execution engine" ("director" in Ptolemy) similar to our "pluggable semantics". Our work addresses different Statechart variants and formal semantics with particular focus on model checking and systematic test case generation, while Ptolemy's goal is simulation.

The parametric semantics from [12–14] provide powerful semantic frameworks for many Statecharts variants as well as process algebras. While quite flexible, they can not fully capture the behavior of any of the three notations considered here (see [15] for details).

2 Design Choices and Extensions

Design Choices. We chose Java as the common language to represent and analyze Statecharts for several reasons. First, we needed an *executable* representation for the models, to allow for quick validation and debugging. Java has a precise, clear semantics, well-understood by many, so implementing a concise simple execution engine for the Statechart variants (that is actually readable) is a good, pragmatic approach to defining semantics. We also wanted a *modular* and *extensible* design for our framework, to allow for easy integration of new semantic variants. Java is an ideal language for this purpose. Furthermore, we chose Java to leverage the model checking and symbolic execution capabilities from JPF for systematic analysis, automated test case generation (with SPF) and coverage measuring.

We also note that the Statechart variants have large action languages. Features like complex data types and function states, along with transitions containing guards and actions that use these types and functions, would be difficult to represent in simpler modeling languages, e.g. satisfiability modulo theories (SMT) formulas that can be solved with off-the-shelf solvers. On the other hand, there is a straightforward mapping from most action-language features into a similar concept in Java.

We have designed the generated code and semantic modules so that they work together to provide a clean input-output interface to the environment. This interface allows us to simulate the models and also to connect them to JPF, with JPF driving the execution non-deterministically or symbolically.

Extensions. The integration of Polyglot with JPF enables us to take advantage of the optimized analysis techniques that are already provided by JPF. To further improve the performance of Statechart analysis in Polyglot, we have experimented with two techniques [16]. The first is a multithreaded custom symbolic execution engine for Polyglot, while the second technique is the application of *partial evaluation* to optimize the generated Polyglot code with respect to particular models and semantics. We note that the design of Polyglot, which decouples the semantic modules from the "structure" of a Statechart model, lends itself well to a multithreaded implementation.

Polyglot can be used as described above to execute and analyze both individual models and also systems with a simple communication that matches Statechart semantics (i.e. event broadcast). This mechanism is insufficient for components that execute in parallel and communicate asynchronously. The problem could be addressed by modeling the communication protocol itself as another Statechart and composing it with the other models. However this may be inefficient, as the protocols can be very large. We have therefore explored extending Polyglot with features not inherent to the basic Statecharts paradigm. These include a connector mechanism for communication and a scheduling framework for sequencing the execution of individual components [17].

Polyglot comes with a library of connectors modeling lossless FIFO communication. Instead of reading data from or sending data directly to another component, data is read from or written to a connector. Other communicating mechanisms, such as lossy communication and non-FIFO message delivery, can be easily incorporated. The scheduler is responsible for ordering the component execution and for invoking the property checking. We have developed a generic scheduler that can be instantiated with different scheduling mechanisms, e.g. non-deterministic, priority-based, calendar-based, etc. By default, Polyglot uses a non-deterministic scheduler. Currently, it is the responsibility of the user to manually link the components via the connector and scheduling mechanism. We intend to automate the process using the Generic Modeling Environment (GME) [18], a graphical tool that already supports our intermediate representation and in which we can describe a system's architecture and automatically generate the code for connector and scheduler instances.

3 Tool Usage

Polyglot has been applied to medium-sized models of flight software, including an example modeling a component from NASA/JPL's Mars Exploration Rovers (MER) [15]. The MER software consists of a Resource Arbiter and several user components, serving specific applications, such as imaging, controlling the robot arm, communicating with earth, and driving. The arbiter moderates access to shared resources, preventing potential conflicts between resource requests and enforcing priorities; e.g., a communication session with Earth can not be started while the rover is driving. Each user has 2 pseudostates, 4 atomic states, 1 compound state and 9 transitions (259 LOC in the Java representation), while the arbiter has 33 pseudostates, 15 atomic states, 2 orthogonal states and 58 transitions (1788 LOC). Polyglot was used for checking safety properties and generating test cases for this model, where the semantics of User 1 was changed from Stateflow into UML and

Table 1. Experimental results

Semantics, Seq. size	Total # Test Cases	Property	Memory, Time
U1 Stateflow, 4	125	true	20 MB, 43 s
U1 Stateflow, 5	412	true	22 MB, 2 m 04 s
U1 Stateflow, 6	1343	true	24 MB, 6 m 46 s
U1 UML, 4	57	false	21 MB, 21 s
U1 UML, 5	155	false	21 MB, 53 s
U1 UML, 6	579	false	23 MB, 2 m 50 s
U1 Rhapsody, 4	57	false	21 MB, 21 s
U1 Rhapsody, 5	155	false	21 MB, 55 s
U1 Rhapsody, 6	579	false	23 MB, 2 m 45 s

Rhapsody. Table 1 shows the results for analyzing the models with increased number of time steps, corresponding to sequences of sizes 4, 5 and 6.

The property holds for the Stateflow models, but it fails when we change the semantics of one user to UML or Rhapsody. This is due to a semantic difference between UML and Stateflow (outer transitions have higher priority over inner transitions in Stateflow, but have lower priority in UML and Rhapsody). This semantic difference is also reflected in the different number of test cases. Note that the results for UML and Rhapsody are practically identical (since their semantic differences are not exposed by the analyzed models).

The feedback produced at the Java-level has the form of test sequences that have been used as inputs to drive the simulation of the models in the original modeling environments. The generated test sequences can also be used for testing the code that is generated from the models.

Polyglot has been used also to analyze models representing the interaction between the Ares launch vehicle and the Orion Crew Exploration Vehicle [17]. The Ares-Orion communication during abort was formulated as a property derived from the official flight software design documents and the software requirements specification available for Ares I. The analysis confirmed problems suspected by the engineer who developed the model, who had already submitted a request for a change to the Ares I design document. Since then, the design has changed to reduce the command echo dependency because of a bit-rate limitation. The effects of that change have not yet been investigated, but our tool can help answer this for the future.

4 Conclusion

We have described Polyglot, a tool for the systematic analysis of model-based software written with multiple Statechart formalisms. The tool has been applied to the analysis of safety-critical systems whose interacting components were modeled using multiple Statechart formalisms. We plan to further expand and robustify the tool and use it for the analysis of the ground system in the GOES-R project [19]. We also plan to explore the compositional techniques from JPF [7] for the component-based analysis of models in Polyglot. As model-driven development is increasingly used in a diverse way for the design and implementation of safety and mission critical systems, we believe that our tool will provide a key capability for the verification and validation of such software.

Acknowledgments. This work has been supported in part by NASA under Cooperative Agreement NNX09AV58A. The authors would also like to thank Michael Whalen and Tom Pressburger for valuable discussions and feedback.

References

1. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3) (June 1987)
2. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Comp. Sci. Dept. Aarhus University, Denmark (1981)
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE (1999)
4. Balasubramanian, D., Pap, G., Nine, H., Karsai, G., Lowry, M.R., Pasareanu, C.S., Pressburger, T.: Rapid property specification and checking for model-based formalisms. In: International Symposium on Rapid System Prototyping (2011)
5. Java pathfinder, `http://babelfish.arc.nasa.gov/trac/jpf`
6. Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of ASE, pp. 179–180 (2010)
7. Giannakopoulou, D., Păsăreanu, C.S.: Interface Generation and Compositional Verification in JavaPathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
8. Polyglot, `https://wiki.isis.vanderbilt.edu/MICTES/index.php/Publications`
9. Hamilton, M.: The heart and soul of apollo: Doing it right the first time. In: Proc. 7th International Military and Aerospace Programmable Logic Devices (MAPLD) Conference (2004)
10. Pezzè, M., Young, M.: Constructing multi-formalism state-space analysis tools: Using rules to specify dynamic semantics of models. In: ICSE (1997)
11. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y.: Taming heterogeneity - the ptolemy approach. Proc. of IEEE 91(1) (2003)
12. Esmaeilsabzali, S., Day, N.A., Atlee, J.M., Niu, J.: Big-step semantics. Technical Report CS-2009-05, David R. Chariton School of Computer Science, Univ. of Waterloo, Ontario, Canada N2l 3G1 (2009)
13. Esmaeilsabzali, S., Day, N.A.: Prescriptive Semantics for Big-Step Modelling Languages. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 158–172. Springer, Heidelberg (2010)
14. Niu, J., Atlee, J.M., Day, N.A.: Template semantics for model-based notations. IEEE Trans. Software Eng. 29(10), 866–882 (2003)
15. Balasubramanian, D., Pasareanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple statechart formalisms. In: ISSTA (2011)

16. Balasubramanian, D., Pasareanu, C.S., Karsai, G., Lowry, M.R., Whalen, M.W.: Improving symbolic execution for statechart formalisms. In: MODEVVA (2012)
17. Balasubramanian, D., Păsăreanu, C.S., Biatek, J., Pressburger, T., Karsai, G., Lowry, M., Whalen, M.W.: Integrating Statechart Components in Polyglot. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 267–272. Springer, Heidelberg (2012)
18. Dubey, A., Karsai, G., Mahadevan, N.: A component model for hard real-time systems: Ccm with arinc-653. Softw., Pract. Exper. 41(12), 1517–1550 (2011)
19. Goes-r, http://www.goes-r.gov

MEMORAX, a Precise and Sound Tool for Automatic Fence Insertion under TSO*

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Carl Leonardsson[1], and Ahmed Rezine[3]

[1] Uppsala University, Sweden
[2] Academia Sinica, Taiwan
[3] Linköping University, Sweden

Abstract. We introduce MEMORAX, a tool for the verification of control state reachability (i.e., safety properties) of concurrent programs manipulating finite range and integer variables and running on top of weak memory models. The verification task is non-trivial as it involves exploring state spaces of arbitrary or even infinite sizes. Even for programs that only manipulate finite range variables, the sizes of the store buffers could grow unboundedly, and hence the state spaces that need to be explored could be of infinite size. In addition, MEMORAX incorporates an interpolation based CEGAR loop to make possible the verification of control state reachability for concurrent programs involving integer variables. The reachability procedure is used to automatically compute possible memory fence placements that guarantee the unreachability of bad control states under TSO. In fact, for programs only involving finite range variables and running on TSO, the fence insertion functionality is complete, i.e., it will find all minimal sets of memory fence placements (minimal in the sense that removing any fence would result in the reachability of the bad control states). This makes MEMORAX the first freely available, open source, push-button verification and fence insertion tool for programs running under TSO with integer variables.

1 Introduction

We introduce MEMORAX, the first freely available, open source (`https://github.com/memorax/memorax`), push-button verification and fence insertion tool that can handle integer variables and that is both sound and complete under TSO for all programs that only involve finite range variables. Modern concurrent processor architectures allow *weak* (*relaxed*) memory models, in which certain memory operations may overtake each other. The use of weak memory models makes reasoning about behaviours of concurrent programs challenging, even for skilled developers. This is for instance witnessed by the lively debate among developers on the Linux Kernel Mailing list about the correctness on x86 of the "Linux Ticket Lock" protocol. (See the mail thread starting with `https://lkml.org/lkml/1999/11/20/76`.) In fact, several synchronisation algorithms, such as mutual exclusion and producer-consumer protocols, turn out to be

* This research was in part funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the National Science Council of Taiwan project no. NSC-101-2221-E-001-007, and the CENIIT research organization (project 12.04).

incorrect if run without modification on weak memories [5]. MEMORAX is based on the techniques developed in [4] and extended in [3]. Not only does our tool turn this verification task into a push-button exercise, it also automatically inserts fences in order to ensure correctness of programs that were made incorrect by the weak memory relaxation. More precisely:

- MEMORAX is an open source [2] push-button tool that comes with a graphical user interface and a simple low level language with a well defined semantics.
- it is sound for concurrent programs running on the TSO memory model (i.e., x86 and SPARC platforms) and involving variables with finite or integer ranges.
- it performs reachability on infinite state spaces to verify control state reachability.
- it provides users with concrete counter-examples, useful for debugging, that take the program from an initial configuration to a specified bad control state.
- it is complete, using an intricate encoding based on the theory of well-quasi-ordering, for the reachability problem of programs on TSO, provided that they only have finite range variables.
- it uses an off-the-shelf SMT solver (MathSAT [1]) to incorporate an interpolation based CEGAR loop to handle integer variables.
- it automatically finds (sets of) fences to ensure a safety property is respected if the property does hold on SC.
- it finds all minimal sets of fences for programs with finite range variables and running on TSO.

Targeted User Base. We see three potential groups of users for MEMORAX :

1. Computer science researchers can use the open source code of MEMORAX to compare with other approaches for the verification of programs running on top of weak memory models, to improve and optimise the implemented techniques (e.g. by interfacing with other SMT solvers or by improving the used data structures or the symbolic representations), or to target new platforms and programs (e.g. add soundness for RMO or PSO, or scale for heap manipulating programs)
2. Teachers of architecture and concurrent programming classes can use (and augment) MEMORAX with its simple user interface in order to familiarise their students with weak memory models. In particular the precision and counter example capabilities of MEMORAX can concretely illustrate the effects of relaxed memory.
3. Software developers working on complex and low level, lock-free code can use MEMORAX to easily check the effects of TSO on their tentative solutions. The generated error traces are also possible on weaker memory models and can conveniently help to highlight possible problems.

Related Tools and Approaches. As far as we know, MEMORAX is the first available open source verification and fence insertion tool that is sound on TSO, that can handle integer variables, and that is complete for programs with finite range variables under TSO. There exists several very conservative approaches that restrict to SC executions by establishing “triangular race freedom” [12] or by inserting fences using “delay set analysis" [13]. We will not further elaborate on those techniques, but will instead focus on a number of tools and approaches more similar to our own.

CheckFence [6] is a SAT-based tool that tests correctness of fence placements by considering finite executions on different relaxed memory models. The tool cannot verify programs that result in buffers of arbitrary size like the ones MEMORAX handles since it unrolls loops and checks correctness of the resulting finite executions.

Fender [8,9] combines model checking with abstraction in order to perform reachability analysis on finite over-approximations. It considers different memory models and uses the reachability analysis to justify fence placements. The analysis is not exact and cannot guarantee to show absence of errors for correct programs. As a result, the tool lacks the precision that would allow it to find minimal sets of fence placements. Unfortunately, we were not able to find the tool which is, as far as we know, not open source. Finally, the tool does not handle programs with integer variables.

mmchecker [7] performs explicit model-checking for the .NET memory model. It explores the (possibly infinite) state space and inserts fences in order to forbid behaviours that are not possible under SC. The tool cannot prove correctness of programs that generate infinite state spaces but do not require fences. Also the tool cannot soundly handle integer variables like MEMORAX does on TSO.

Automata based accelerations [10,11] computes under-approximations of the generated infinite state space on different relaxed memory models. When the analysis terminates, it answers exactly whether the property is violated or not, and it allows to deduce minimal sets of fence placements, even for programs that may generate buffers of arbitrary sizes. The approach targets systems that manipulate finite variables. It neither can handle integer variables nor does it guarantee termination. We were not able to get hold of the tool or of its source code.

```
forbidden
  CS CS

data
  turn = * : [0:1]
  x    = 0 : [0:1]
  y    = 0 : [0:1]

process
registers
  $r0 = * : [0:1]
  $r1 = * : [0:1]
text
  L0: write: x := 1;
  write: turn := 1;
  L1: read: $r0 := y;
  read: $r1 := turn;
  if $r0 = 1 && $r1 = 1 then
    goto L1;
  CS: write: x := 0;
  goto L0

process
registers
  $r0 = * : [0:1]
  $r1 = * : [0:1]
text
  L0: write: y := 1;
  write: turn := 0;
  L1: read: $r0 := x;
  read: $r1 := turn;
  if $r0 = 1 && $r1 = 0 then
    goto L1;
  CS: write: y := 0;
  goto L0
```

Fig. 1. Peterson's mutual exclusion protocol

2 Using the Tool

2.1 The RMM Language

Programs to be tested with MEMORAX are written in the special purpose language RMM. For reasoning about programs under relaxed memory, detailed knowledge about how variables are stored and used is necessary. RMM is designed to unambiguously describe that aspect by making memory accesses and register use explicit.

As an example, Figure 1 shows an RMM model of the Peterson mutual exclusion protocol. Lines 1-2 are of particular interest, since they specify the safety criterion: It is forbidden for the processes (henceforth called `P0` and `P1`) to simultaneously be in the control states labelled `CS` (i.e. line 20 for `P0` and line 34 for `P1`).

2.2 Usage through the Graphical Interface

The GUI is a python script (`memorax-gui`) wrapping around the CLI. The GUI window consists of three main parts: The command input area, the code area and the output area. The command input area provides the commands "Reachability", "Fence insertion" and "Draw automata", and options for the commands. All commands apply to the code in the code area, and print their output (and possibly errors) to the output area.

A typical work flow would be the following: First write the RMM code for the protocol you want to analyse. Then use the "Draw automata" command to produce a PDF file showing the automata for the defined processes. This is useful for asserting that the RMM code specifies what you intended. Next use the "Reachability" command to check whether the protocol is safe from the start. If not, then use the "Fence insertion" command to receive sets of fences that will make the protocol safe.

Reachability. The Reachability command is used to analyse whether there is some configuration which violates the safety specification, but is reachable from some initial configuration. If there is such a configuration, then an error trace will be supplied.

There are currently two reachability methods ("abstractions") available in MEMORAX: SB ("Single Buffer") and PB ("Predicate abstraction and buffer Bounding"), corresponding respectively to our works in [4] and [3]. The PB method is an over-approximation and allows for CEGAR abstraction refinement.

Protocols can be automatically rewritten to "*Register Free Form*" before being analysed. This encodes register values in control states, and can often improve analysis performance.

Fence Insertion. The fence insertion command will repeatedly execute reachability queries, while gradually adding fences to the analysed protocol in order to guarantee satisfaction of the safety criterion. The available options for fence insertion are the same as for reachability, and apply to the repeated reachability queries.

Interpreting the Output: If we apply the fence insertion command to the program in Figure 1, we will get output describing the results of the reachability queries. There will be a description of the result at the end of the output:

```
Found 1 fence set:
Fence set #0:
  L15 P0: write: turn := 1
  L29 P1: write: turn := 0
```

Here MEMORAX has found exactly one minimal and sufficient set of fences, namely the one corresponding to locking the writes at line 15 and 29. Other possible outcomes include the empty set - meaning the program is already correct, and no sets - meaning the program cannot be corrected with fences.

3 Implementation

MEMORAX is implemented in C++ with the intent of being easy to extend with new memory models and analysis methods.

Reachability Optimisations. We mention some of the techniques we use to combat the state space explosion problem:

- *Light-Weight Pre-Analysis.* Before the reachability analysis is started, we apply a light-weight, per-thread, over-approximating analysis. This allows us to collect a rough invariant about the buffer contents that are possible per control state and process. We use the invariant to efficiently reduce the explored state space.
- *Update Restriction.* We soundly limit store buffer updating to only take place after a read instruction by the same process. The rationale is that it is only relevant to delay a write instruction by buffering if it is delayed past a read instruction. Other delays can be simulated under SC.
- *Partial Order Reduction for TSO.* In addition to the above update limitation, MEMORAX uses a partial order reduction technique based on the principle that an instruction reordering that does not participate in a conflict cycle, as defined in [13], can be simulated by an appropriate scheduling under SC. Thus instruction reorderings that do not participate in conflict cycles need not be analysed.

Fence Insertion. The fence insertion algorithm relies on the underlying reachability analysis when evaluating each fence set placement. It is therefore desirable to keep the number of tried fence sets as small as possible.

- *Fence Placement Restriction.* We restrict the number of possibilities, by only considering fences that can be added by locking some write instruction. For example, changing `write: x := 1` into `locked write: x := 1` adds a fence after `write: x := 1`. This guarantees finding minimal and sufficient fence sets (if they exist). Their size can however be larger than a smallest sufficient set.
- *Multiple Fence Extraction.* We perform an extensive analysis to capture fences that need to be added in order to avoid a given error trace. By identifying the conflict cycles (as described by [13]) that a particular reordering $(a \to b)$ participates in, it is sometimes possible to deduce the existence of another, similar error trace where a and b occur in program order, but another pair $(c \to d)$ is reordered, yielding the same conflict cycle. In such cases a fence between c and d is equally necessary as a fence between a and b. Thus the fence insertion algorithm can infer more than one fence at a time, and the number of reachability queries can be decreased.

4 Experimental Results

Table 1 displays the results of running MEMORAX on several classical examples. For each of the examples, we give the total time for finding all minimal, sufficient sets of fences, using the methods SB and PB, with and without transforming the program to register free form. In the table, "not-applicable" denotes that the corresponding approach is not applicable to the example. These correspond to applying a finite domain technique (SB and RFF) to an infinite domain program. Furthermore, out-of-mem is used to denote that the experiment failed to finish before consuming all available memory of the host computer. All examples were run on a laptop with a 2.27 GHz processor and 4 GB of memory.

Table 1. Experimental Results

	Size Proc./States/ Var./Trans.	Total time seconds				Fences necessary (smallest set)
		SB	SB(rff)	PB	PB(rff)	
Simple Dekker	2/6/2/6	0.0	0.0	0.0	0.0	1 per proc
Full Dekker	2/22/3/28	0.4	0.2	0.1	0.1	1 per proc
Peterson	2/12/3/14	1.9	1.0	3.5	0.4	1 per proc
Lamport Bakery (bounded)	2/18/4/20	out-of-mem	61.2	152.7	17.9	2 per proc
Lamport Fast	2/24/4/34	233.7	223.4	2.7	2.5	2 per proc
CLH Queue Lock	2/30/4/42	out-of-mem	15.4	out-of-mem	out-of-mem	0
Sense Reversing Barrier	2/4/2/4	0.3	0.2	0.1	0.0	0
Burns	2/8/2/9	0.0	0.0	0.0	0.0	1 per proc
Dijkstra	2/22/3/28	out-of-mem	0.4	1.0	2.0	1 per proc
Lamport Bakery (unbounded)	2/18/4/20	not-applicable	not-applicable	166.2	not-applicable	2 per proc
Linux Ticket Lock (unbounded)	2/4/2/4	not-applicable	not-applicable	0.4	not-applicable	0

References

1. MathSAT4, http://mathsat4.disi.unitn.it/
2. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: https://github.com/memorax/memorax
3. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Automatic Fence Insertion in Integer Programs via Predicate Abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 164–180. Springer, Heidelberg (2012)
4. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-Example Guided Fence Insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
5. Adve, S., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer 29(12) (1996)
6. Burckhardt, S., Alur, R., Martin, M.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
7. Huynh, T.Q., Roychoudhury, A.: A Memory Model Sensitive Checker for C#. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 476–491. Springer, Heidelberg (2006)
8. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2011)
9. Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI (2011)
10. Linden, A., Wolper, P.: An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)

11. Linden, A., Wolper, P.: A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 144–160. Springer, Heidelberg (2011)
12. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
13. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. Transactions on Programming Languages and Systems 10, 282–312 (1988)

BULL: A Library for Learning Algorithms of Boolean Functions[*]

Yu-Fang Chen and Bow-Yaw Wang

Academia Sinica, Taiwan
http://bull.iis.sinica.edu.tw/

Abstract. We present the tool BULL (Boolean fUnction Learning Library), the first publicly available implementation of learning algorithms for Boolean functions. The tool is implemented in C with interfaces to C++, JAVA and OCAML. Experimental results show significant advantages of Boolean function learning algorithms over all variants of the L^* learning algorithm for regular languages.

1 Introduction

BULL is the first publicly available implementation of learning algorithms for Boolean functions. Three algorithms are implemented in the library. The classical CDNF algorithm infers Boolean functions over a fixed number of variables. The incremental CDNF+ and CDNF++ algorithms infers Boolean functions over an indefinitely number of variables. The library is implemented in C with C++, JAVA, and OCAML interfaces. Sample codes of C, C++, JAVA, and OCAML are distributed with the library. Users can adopt BULL by modifying them.

What Is It?. Learning algorithms for Boolean functions can be viewed as an efficient procedure to generate a *target* Boolean function only known to a teacher. This type of learning algorithms assume a teacher who answers queries about the target Boolean function. The learning algorithms acquire information from the answers to queries and organize them in a systematic way. In the worst case, learning algorithms will infer a target Boolean function within a polynomial number of queries in the CNF and DNF formula sizes of the target function.

Learning in Formal Verification. Since the work in [8], algorithmic learning has been applied to formal verification techniques such as specification synthesis [8], automated compositional verification [5], and regular model checking [6]. Most applications are based on the L^* automata learning algorithm for regular languages. The learning algorithm enumerates states explicitly. Its applications are hence inherently explicit [5], or use explicit automata as implicit representations of state spaces [6].

[*] The project was supported in part by the National Science Council of Taiwan project no. NSC-101-2221-E-001-006 and no. NSC-101-2221-E-001-007.

Why Use Boolean Learning. Implicit algorithms (e.g., SAT-based model checking) can greatly improve the capacity of various verification techniques. Similar improvements have also been reported in applications of the CDNF learning algorithm for Boolean functions. In [3], the learning algorithm is adopted to infer implicit contextual assumptions in automated compositional reasoning. It is shown that learning implicitly can tackle certain hard problems unattainable by traditional explicit algorithms. The CDNF algorithm is also applied to loop invariant generation. The learning-based framework can be much more efficient than conventional static analysis algorithms [7].

For regular languages, the learning algorithms are available in veteran tools (such as libalf [1] and learnlib [11,10,9]). Implementations of learning algorithms for Boolean functions however are still missing. Since it would take a considerable amount of time to understand and implement learning algorithms for Boolean functions, lack of publicly available tools could be an obstacle to develop related techniques in the research community. In order to lower the barrier to entry, we decide to develop the BULL library.

The Position of the Paper. The Boolean learning project starts in 2009 and since then we tested different variants of the algorithms and data structures. Several of them indeed dramatically improved the performance, e.g., non-membership queries are introduced partly for performance reasons. However, since boolean learning is a new technique to most people in the community. We decided to spend the pages for a general introduction instead of technical details.

2 The BULL Library

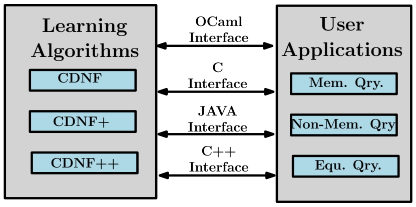


Fig. 1. System Architecture

Figure 1 shows the architecture of the BULL library. The core library contains three learning algorithms implemented in C. They are the CDNF [2], the CDNF+ [4], and the CDNF++ [4] algorithms. The CDNF algorithm assumes that the number of variables in the target Boolean function is known. The CDNF+ and the CDNF++ algorithms do not have this assumption. In addition to the learning algorithms, we also provide C++, JAVA (via JNI), and OCaml interfaces.

2.1 How to Use the Package

In order to adopt the learning algorithms in BULL, users have to play the teacher and answer queries posed by the algorithms. For the sake of presentation, let us assume that $f(x, y, z) = (x \wedge \neg y) \vee (x \wedge z)$ is the target Boolean function over variables x, y, and z. Consider the following sample queries from the learning algorithms:

1. A *membership* query on a partial assignment $\{(x, \mathit{false})\}$. On a membership query, the teacher checks if the target is satisfiable under the given assignment. Here the teacher answers *no* since $f(\mathit{false}, y, z)$ is not satisfiable.
2. A *non-membership* query on a parital assignment $\{(y, \mathit{true})\}$. On a non-membership query, the teacher checks if the negation of the target is satisfiable under the assignment. For this example, the teacher answers *yes*.
3. An *equivalence* query on a conjecture $f'(x, y, z) = x \wedge y$. On an equivalence query, the teacher answer *yes* if the given formula is equivalent to the target. Otherwise, she returns an assignment as a counterexample. For this example, the teacher may return the assignment $\{(x, \mathit{true}), (y, \mathit{true}), (z, \mathit{false})\}$ since $f'(\mathit{true}, \mathit{true}, \mathit{false}) \neq f(\mathit{true}, \mathit{true}, \mathit{false})$.

Table 1. Features of Algorithms

	Num. of Vars.	Mem. Qry,	Non-Mem. Qry.	Equ. Qry.
CDNF	known	√		√
CDNF+	unknown	√		√
CDNF++	unknown	√	√	√

Different learning algorithms pose different types of queries. Table 1 shows the differences among the three learning algorithms in BULL. The CDNF algorithm assumes the number of variables in the target Boolean function is known. The CDNF+ algorithm does not know the number of variables. Both algorithms only pose membership and equivalence queries. The CDNF++ algorithm does not presume the number of variables is known. It however poses membership, non-memberhip, and equivalence queries.

BULL defines the interfaces to the three types of queries. If all queries can be answered automatically, users can implement a mechanical teacher to answer queries through the interface. Learning algorithms in BULL will invoke the mechanical teacher and infer unknown target functions automatically. We refer interested users to our full version (`http://bull.iis.sinica.edu.tw/`) which contains a detailed demonstration of how to implement the above query functions and connect them to BULL.

2.2 Users of BULL

The BULL library targets the formal verification research community. As far as we know, several people in the field are interested in the applications of learning algorithms for Boolean functions. The library has already been used by the verification group in Oxford University (Learning-based Compositional Probabilistic Model Checking), the software trustability and verification group in Tsinghua University (Learning-Based Compositional Verification), and the static analysis group in Seoul National University (Loop Invariant Inference). Several other groups have shown their interests and asked for the source code.

2.3 Potential Applications

The CDNF algorithm has been applied to synthesize contextual assumptions in assume-guarantee reasoning. It has also been used to infer a loop invariant in program verification. These applications share common characteristics. First, computing contextual assumptions or loop invariants without learning is possible but expensive. It is however easy to verify if purported contextual assumptions or loop invariants work. Moreover, contextual assumptions or loop invariants are by no mean unique. It suffices to compute but one contextual assumption or loop invariant in these applications. From our experience, we believe that learning is most suitable for problems with the aforementioned characteristics.

For interested reader, a step-by-step tutorial of how to use the BULL library to find loop invariants is provided in our full version (`http://bull.iis.sinica.edu.tw/`). We hope it may give some insights to more applications of the library.

3 Experimental Results

Since the target application of BULL is verification, in the first experiment, we decide to pick a classical example, n-bit counter, as the target for learning (Table 2). In Table 3, we show a different version where the n-bit counter model can be non-deterministically reset to 0 from any state. In the second experiment, we compare the performance of the Boolean learning algorithms using random 3SAT formulae of n variables. In those formulae, the ratio of the number of variables to the number of clauses is 1/4.[1] We use a timeout period of 10 minutes. In Figure 2, we show the average execution time of the first 50 non-trivial instances (satisfiable and all algorithms finished within the timeout period). In Table 4, we show the number of timeout cases out of 180 instances.

Table 2. Comparison of Boolean function learning algorithms: using n-bit counter as the example

	2	3	4	5	6	7	8	9	10	11	12
CDNF	0.02	0.02	0.05	0.11	0.35	1.03	2.29	4.3	9.8	23.6	66.2
CDNF+	0.01	0.02	0.04	0.09	0.27	0.77	1.5	2.4	5.7	14.1	40.3
CDNF++	0.01	0.02	0.04	0.09	0.25	0.77	1.5	2.4	5.6	13.8	39.8

At the first glance, CDNF learning algorithm has the best performance among the three. However, it is not a fair interpretation for two reasons. First, CDNF makes use of some information (number of variables in the target function) that is not known by the other two algorithms. More importantly, in particular for

[1] This ratio is very close to satisfiability threshold of 3SAT formulae. Hence the chance of getting a satisfiable formula is 50%.

Table 3. Comparison of Boolean function learning algorithms: using n-bit counter with non-deterministic reset as the example

	2	3	4	5	6	7	8	9	10	11	12
CDNF	0.00	0.02	0.07	0.24	0.75	2.83	12.13	32.01	112	451	1374
CDNF+	0.01	0.02	0.06	0.21	0.67	2.63	12.1	36.8	144	637	1671
CDNF++	0.01	0.02	0.06	0.21	0.62	2.63	12.08	36.88	145	582	1632

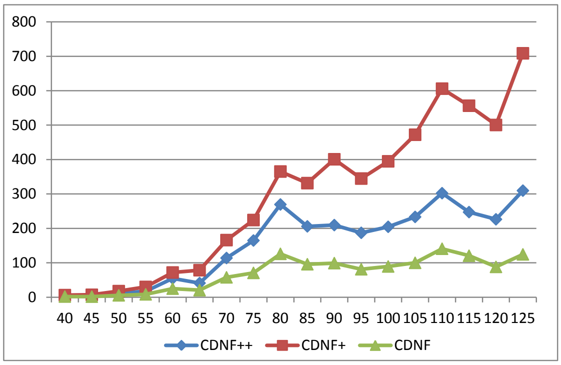


Fig. 2. Comparison of Boolean learning algorithms, using random 3SAT formulae as the benchmark. The vertical axis is the average execution time in seconds and the horizontal axis is the number of variables in the formula. Each point is the average results of 50 instances.

the case of randomly generated formulae, almost all the variables will be added to the final result. Hence the benefit obtained from incremental learning is not significant in such type of examples. In fact, the CDNF+ and CDNF++ algorithms are particularly useful in formal verification applications [4] such as those based on predicate abstraction and interpolation-based refinement. Typically in these applications, a boolean variable is used to indicate the truth of a predicate in certain points of program executions. Since the number of predicates in use would increase in each refinement step, there is no *a prior* known upper bound of needed variables.

Table 4. The number of timeout cases out of 180 instances

Num. of Var.	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120	125	130	135
CDNF	0	0	0	0	0	0	1	2	0	14	7	24	28	31	48	51	70	76	90	93
CDNF+	0	0	0	0	0	0	1	6	5	21	19	42	48	51	83	80	88	99	118	122
CDNF++	0	0	0	0	0	0	2	4	6	19	16	32	40	45	69	69	82	90	106	109

References

1. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: `libalf`: The Automata Learning Framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010)
2. Bshouty, N.H.: Exact learning Boolean function via the monotone theory. Information and Computation 123(1), 146–153 (1995)
3. 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)
4. 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)
5. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning Assumptions for Compositional Verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
6. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. In: ENTCS, pp. 21–36 (2005)
7. 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)
8. Maler, O., Pnueli, A.: On the learnability of infinitary regular sets. Information and Computation 118(2), 316–326 (1995)
9. Merten, M., Howar, F., Steffen, B., Cassel, S., Jonsson, B.: Demonstrating Learning of Register Automata. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 466–471. Springer, Heidelberg (2012)
10. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next Generation LearnLib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011)
11. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models. STTT 11(5), 393–407 (2009)

AppGuard – Enforcing User Requirements on Android Apps

Michael Backes[1,2], Sebastian Gerling[1], Christian Hammer[1], Matteo Maffei[1], and Philipp von Styp-Rekowsky[1]

[1] Saarland University, Saarbrücken, Germany
[2] Max Planck Institute for Software Systems (MPI-SWS)

Abstract. The success of Android phones makes them a prominent target for malicious software, in particular since the Android permission system turned out to be inadequate to protect the user against security and privacy threats. This work presents *AppGuard,* a powerful and flexible system for the enforcement of user-customizable security policies on untrusted Android applications. AppGuard does not require any changes to a smartphone's firmware or root access. Our system offers complete mediation of security-relevant methods based on callee-site inline reference monitoring. We demonstrate the general applicability of AppGuard by several case studies, e.g., removing permissions from overly curious apps as well as defending against several recent real-world attacks on Android phones. Our technique exhibits very little space and runtime overhead. AppGuard is publicly available, has been invited to the Samsung Apps market, and has had more than 500,000 downloads so far.

1 Introduction

Mobile devices nowadays store a plethora of sensitive information about us – both private and business-related. Usually, this information can be accessed in predefined locations, such as address books or photo folders, and is thus easily locatable by an attacker. Most of these locations, however, lack comprehensive access control and protection mechanisms. When users install a new app on Android, they have no choice but to grant an app all requested permissions at install time, and these permissions cannot be revoked later on. At the same time, these permissions are coarse-grained and their impact is hard to understand for the average user. In the past, several incidents have been reported where private information was deliberately leaked to external servers. Even widely used major apps like Twitter and WhatsApp used to clandestinely send the phone's whole address book to their servers to mine for possible contacts (for iOS, similar behavior was revealed, e.g., for the Facebook app).

In order to overcome this unsatisfactory situation, this paper presents AppGuard, a tool based on inline reference monitoring (IRM) [4,3] that allows the user to enforce fine-grained security and privacy policies on third-party apps. These policies enforced by AppGuard restrict the outreach of vulnerabilities both in third-party applications and the operating system. In short, the IRM algorithm proceeds in two steps. First, app binaries are rewritten to invoke at runtime a security monitor before each security-relevant program operation, usually before each function call to the Android system libraries. Second, the security monitor dynamically checks whether any of the currently enforced security policies allows the attempted operation, and then either grants the execution or executes alternative code (e.g., to return a mock value to prevent the app's termination due to an exception). Since IRM only affects the app binary and not the operating system, AppGuard allows for enforcing policies without rooting phones or changing the operating system.

AppGuard is deployed as a stand-alone app, has been installed on about 500,000 phones so far, and will be soon released to the Samsung Apps market after an explicit invitation from Samsung. The experimental evaluation and case studies discussed in this paper demonstrate the effectiveness of our approach: AppGuard exhibits very little overhead in terms of space and runtime, and it can be used to revoke permissions of excessively curious apps, to enforce complex policies, and to prevent several recent real-world attacks on Android phones.

Although several approaches for enforcing policies in Android based on IRM have recently been presented in the literature [7,2], AppGuard is the only IRM based security tool that has been deployed on a large scale and provides a fully automated on-the-phone instrumentation for third-party apps. In the remainder of this paper, we focus on the architecture and on usability and deployment aspects of the tool: for more details on the IRM algorithm and an extensive discussion of the related work, we refer to [1].

2 AppGuard

Architecture. AppGuard uses caller-site rewriting to inline the reference monitor into existing third-party apps. Fig. 1 provides an overview of its components.

Policies. AppGuard provides a set of built-in security and privacy policies. The tool, in particular, provides general purpose policies that aim at the revocation and restriction of critical Android permissions, such as the Internet-, Contacts- and SendSMS-permission. The Internet policy, for example, provides, besides a general on/off switch option, the possibility to specify a set of servers an app is allowed to connect to. The current version of AppGuard contains 24 different policies in total. Security policies are specified in an aspect-oriented programming style and include a detailed specification of all function calls that are to be controlled by the security monitor (cf. Section 3).

Rewriter. Policies constitute the working basis for the rewriting component to inline the specified checks in front of function calls. The rewriter takes an existing application package (`.apk` file), extracts the `classes.dex` file, and disassembles it. After analyzing the converted assembly code, the rewriter merges the security checks specified by the policy into the existing application code. Finally, it reassembles the `classes.dex` file and repackages the apk file. Our implementation handles both reflective JAVA calls and virtual methods.

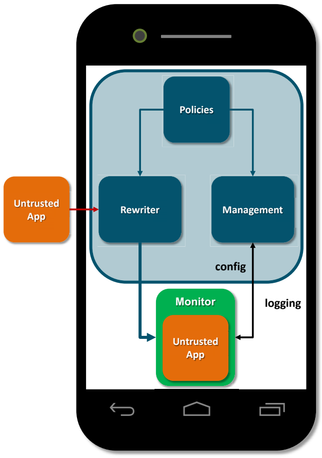
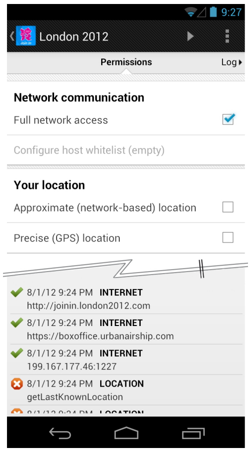


Fig. 1. Architecture of AppGuard



Network communication

Full network access [✓]

Configure host whitelist (empty)

Your location

Approximate (network-based) location ☐

Precise (GPS) location ☐

```
✓ 8/1/12 9:24 PM  INTERNET
  http://joinin.london2012.com
✓ 8/1/12 9:24 PM  INTERNET
  https://boxoffice.urbanairship.com
✓ 8/1/12 9:24 PM  INTERNET
  199.167.177.46:1227
✗ 8/1/12 9:24 PM  LOCATION
  getLastKnownLocation
```

Fig. 2. Permission revocation policies (upper part) and the event log (lower part)

Management. Our management component offers the possibility to select a set of predefined policies and to switch single policies on/off on the fly.

Monitor. The monitor is responsible for the actual enforcement of security policies. It tracks the state of the program execution and decides based on the policy configuration whether a security-relevant operation is allowed or not.

Deployment. A major design decision for AppGuard was its development as a standalone app. This is a crucial requirement for a broad deployment on existing smartphones, since the average user of smartphones is not able or willing to "root" the smartphone or to modify the operating system. As Android enforces app isolation by running every app in its own dedicated sandbox, there is no direct possibility to modify the code of other apps, which, however, is required for inline reference monitoring. We solve this problem by leveraging the fact that Android stores app packages in a world-readable location of the filesystem. Thereby, AppGuard can read the `.apk` packages of installed apps and start the rewriting process. In order to install the modified (secured) app, the user is asked to uninstall the original app and to confirm the installation of the secured app instead. This is due to the fact that, for security reasons, Android does not allow apps to silently uninstall other apps.

Since our rewriting process modifies the original app package, the package signature becomes invalid. Therefore, we have to re-sign the secured application with a new key (usually one key per app developer) such that the original app behavior is preserved. For example, Android makes it possible for apps signed with the same key to access each other' s data: the signing mechanism implemented in AppGuard preserves this behavior.

Usability. AppGuard is designed for ease-of-use and does not require any specific security knowledge. In the following, we briefly outline the typical workflow experienced by the user. Whenever a new app is installed on the phone, AppGuard prompts the user to secure the new app. Clicking the notification takes the user to the initial screen for the app instrumentation, which explains the 3-step rewriting process: (*i*) scanning and rewriting of the target app, (*ii*) uninstallation of the original app, and (*iii*) installation of the modified app. Once the modified app is installed, AppGuard allows the user to grant or revoke individual permissions and configure predefined security policies. For example, the user can specify which hosts the app is allowed to connect to. Furthermore, AppGuard keeps a log of all security-relevant operations performed by an app, providing insights into the app behavior and enabling the user to make informed decisions about the policy configuration. Finally, AppGuard includes an on-the-phone manual and gives an overview of installed and secured apps.

Current Release. AppGuard is implemented as a stand-alone app for Android and is purely written in Java. Our tool was first released in July 2012 and its current code base consists of approx. 6500 lines of code. It builds upon the *dexlib*[1] library, which is used for manipulating App binaries (`dex` files). A prototype of AppGuard was originally implemented by researchers at Saarland University [1]. The currently deployed version is based on that work, and it has been built and is maintained by a spin-off company called Backes SRT. The app has been evaluated on a number of real world applications from Google's official app market Google Play (cf. Section 3 for details on some of our case studies).

AppGuard is available for free and supports all Android versions starting from Android 3.0. The application binary can be downloaded from several websites[2]. It has achieved within a few months a large user basis, especially in Europe. Since its first release, it received significant attention in the German media (e.g., a report within ARD Tagesschau, a news transmission of the first German TV channel). The current version has been downloaded more than 500,000 times and the downloads are increasing steadily. Recently, we have been invited to put AppGuard into Samsung's Apps market where Samsung maintains selected apps especially for their own smartphones.

Upcoming Release. Besides the previously described functionalities, the upcoming release of AppGuard implements a wider range of policies (e.g., for redirecting HTTP connections to HTTPS and for preventing `Runtime.Exec()` calls, which are commonly used in recent Android malware). Further, the new release takes care of updating secured apps as the previously described re-signing procedure renders the updates within Google Play impossible.

Limitations. AppGuard monitors both direct Java calls and calls from native code to Java methods. However, it does not monitor function calls inside of native libraries. In case native code is present, it informs the user and asks whether native code should be executed. According to Zhou et al. [8], only less than 5% of all apps include native libraries.

[1] Part of the *smali* disassembler for Android by Ben Gruver [6]

[2] `http://www.chip.de/downloads/SRT-AppGuard-Android-App_56552141.html`
`http://www.heise.de/download/srt-appguard-1187469.html`

Table 1. Inliner evaluation: sizes of apk file, classes.dex, inlined classes.dex, diff. of dex file, # of total and changed instructions, inlining time on the phone

App (Version)	**Size** [Kb]				**Instructions**		**Time** [sec]
	Apk	**Dex**	**Inl**	**Diff**	**Total**	**Chg**	**Phone**
Angry Birds (2.0.2)	15018	994	1038	+44	79311	100	43.4
Endomondo (7.0.2)	3263	1635	1680	+45	134452	88	23.0
Facebook (1.8.3)	4013	2695	2744	+48	224285	218	47.3
PPXIU (1.0) (infected w/ YZHCSMS)	856	793	839	+46	114427	120	19.9
Super Guitar Solo (1.0.1) (infected w/ DroidDream)	1617	120	161	+41	8641	18	4.5
Twitter (3.0.1)	2218	764	813	+48	105594	107	16.7
Wetter.com (1.3.1)	4296	958	1000	+43	89655	36	15.7

3 Performance Evaluation and Case Studies

This section presents the results of the performance evaluation that we conducted on a Google Galaxy Nexus smartphone (1.2 GHz CPU, 1GB RAM) with Android 4.0.4. and discusses some of the case studies conducted with AppGuard.

Table 1 provides statistics on inlining a representative set of apps with AppGuard. We observed a negligible increase in filesize and reasonable inlining times. Besides, we evaluated the runtime overhead introduced by AppGuard through micro-benchmarks (cf. Table 2). The overhead varies depending on the measured function call, but overall we did not recognize any noticeable slowdown. Performance critical apps like games and video players are usually not negatively affected by this slowdown since most time critical computations are performed in native libraries where no security critical information is involved. Our studies indicate that monitored API functions are not frequently called (e.g. in loops).

We evaluated AppGuard in several case studies based on real world applications and successfully enforced different classes of security and privacy policies. Let us consider as an example the *Twitter* app that used to upload the user's address book to the Twitter servers without user consent: revoking the Contacts permission preserves all major functionalities (the find friends function is, of course, limited), but it prevents the privacy leak. Following the same approach AppGuard can also successfully curb the impact of malware. The YZHCSMS malware, for example, sends SMS to premium numbers, which can be prevented by revoking the SendSMS permission. By means of the *Wetter.com* app, we demonstrate the enforcement of a more fine-grained policy by only allowing connections to the

Table 2. Runtime comparison with micro-benchmarks for function calls in unmodified apps and inlined apps with policies disabled and enabled. The runtime overhead is presented for the inlined app with disabled policies.

Function Call	**Original App**	**Inlined App**		**Overhead**
		Pol. disabled	**Pol. enabled**	
`Socket-><init>()`	0.2879 ms	0.3022 ms	0.0248 ms	5.0%
`ContentResolver->query()`	10.484 ms	11.138 ms	0.1 ms	6.2%
`Camera->open()`	150.8 ms	152.36 ms	0.6 ms	1.0%

wetter.com servers, which are used to retrieve the current weather forecast. By blocking all other network connections, the app no longer displays in-app advertisements. Furthermore, AppGuard is able to mitigate weaknesses both in third party apps and in the OS itself. The *Endomondo Sports Tracker*, for example, leaks the authentication token via unsecured HTTP connections. AppGuard can successfully prevent this leakage by enforcing the usage of HTTPS (if supported). An example of an OS vulnerability in Android is the lack of access control mechanisms for the Android photo storage, which is demonstrated by the proof-of-concept exploit implemented in the *(Evil)Tea Timer* app [5]. AppGuard successfully fixes this vulnerability by allowing the user to control the access to her private photos. Finally, many malware authors try to use binary root exploits to gain elevated privileges (e.g. DroidDream). By monitoring API-calls like `Runtime.exec()`, AppGuard can also prevent this type of attacks.

4 Conclusion

This work presents *AppGuard,* a powerful and flexible system for the enforcement of user-defined security policies on untrusted Android applications. AppGuard is based on IRM and does not require any changes to a smartphone's firmware or root access. We demonstrated the feasibility of our approach through an experimental evaluation and several case studies. We take the size of the current user basis of AppGuard as an indication that it tackles a pressing need on Android.

Acknowledgement. This work was supported by the initiative for excellence, the Emmy Noether program of the German research foundation (DFG) and the German Federal Ministry of Education and Research (BMBF) within the Center for IT-Security, Privacy and Accountability (CISPA) at Saarland University.

References

1. Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P.: Appguard - real-time policy enforcement for third-party applications. Tech. Rep. A/02/2012, Saarland University, Computer Science (July 2012)
2. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-ARM-Droid: A rewriting framework for in-app reference monitors for android applications. In: Mobile Security Technologies 2012, MoST 2012 (2012)
3. Erlingsson, Ú.: The Inlined Reference Monitor Approach to Security Policy Enforcement. Ph.D. thesis, Cornell University (January 2004)
4. Erlingsson, Ú., Schneider, F.B.: IRM enforcement of java stack inspection. In: Proc. 2002 IEEE Symposium on Security and Privacy (Oakland 2002), pp. 246–255 (2002)
5. Gootee, R.: Evil tea timer (2012), `https://github.com/ralphleon/EvilTeaTimer`
6. Gruver, B.: Smali: A assembler/disassembler for android's dex format
7. Jeon, J., Micinski, K.K., Vaughan, J., Fogel, A., Reddy, N., Foster, J., Millstein, T.: Dr. Android and Mr. Hide: Fine-grained permissions in android applications. In: ACM CCS Works. on Sec. & Privacy in Smartphones and Mobile Devices (2012)
8. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In: Proc. NDSS 2012 (February 2012)

Model Checking Database Applications

Milos Gligoric[1] and Rupak Majumdar[2]

[1] University of Illinois at Urbana-Champaign, USA
[2] Max Planck Institute for Software Systems, Germany
gliga@illinois.edu, rupak@mpi-sws.or

Abstract. We describe the design of DPF, an explicit-state model checker for database-backed web applications. DPF interposes between the program and the database layer, and precisely tracks the effects of queries made to the database. We experimentally explore several implementation choices for the model checker: stateful vs. stateless search, state storage and backtracking strategies, and dynamic partial-order reduction. In particular, we define independence relations at different granularity levels of the database (at the database, relation, record, attribute, or cell level), and show the effectiveness of dynamic partial-order reduction based on these relations.

We apply DPF to look for atomicity violations in web applications. Web applications maintain shared state in databases, and typically there are relatively few database accesses for each request. This implies concurrent interactions are limited to relatively few and well-defined points, enabling our model checker to scale. We explore the performance implications of various design choices and demonstrate the effectiveness of DPF on a set of Java benchmarks. Our model checker was able to find new concurrency bugs in two open-source web applications, including in a standard example distributed with the Spring framework.

1 Introduction

We present the design, implementation, and evaluation of DPF, an explicit-state model checker for database-backed web applications. Most web applications are organized in a three-tier architecture consisting of a presentation tier, a business-logic tier, and a persistent database tier. In a typical usage scenario, a user of the application starts a session, makes one or more requests to the application, and then closes the session. The processing of a request depends on the state of the database and can modify the database. The application server hosting the web application assigns a new thread for each request, and runs the logic implemented in the business tier to handle the request. The thread may access the data tier to store or retrieve information to/from a database. After each request is handled, the response is sent back to the user. Since the underlying http protocol is stateless, the application threads typically do not share the state directly. Instead, all state is stored in the session object and in the persistent store. In particular, requests by different users (or requests made in different sessions) only share the database and no other shared state. Most modern languages provide frameworks that simplify the development of web applications (e.g., Spring and Grails for Java, Django for Python, Rails for Ruby).

Since web applications concurrently process requests made by multiple users, there is the potential for concurrency bugs. One class of bugs are atomicity violations, where the application may perform several sequential interactions with the database, without ensuring the sequence occurs atomically. This can expose inconsistent states to the database. Consider two users getting a record and then concurrently deleting it; the second attempt to delete may fail. Note that such bugs can arise even when the implementation of the database management system (DBMS) correctly implements ACID semantics for each transaction. Existing techniques for checking race conditions and atomicity violations [7, 26] in multi-threaded code cannot be applied directly, as they usually depend on explicit tracking of synchronization operations or on heap cells that are read or written.

Since bugs in web applications can have high financial costs, it is reasonable to consider developing systematic state-space exploration tools that look for property violations. Moreover, the application domain makes model checking [2, 12, 14] especially attractive: each request handler typically makes few interactions with the database, to improve latency of the application. Thus, techniques such as partial-order reduction are expected to work exceptionally well: each thread can atomically run all its code between two database transactions. At the same time, developing such a model checker presents new and non-trivial technical challenges: How can we represent the state of the database in the model checker? How can we store and restore states, particularly in the presence of a large amount of data in the database? How do we perform partial-order reduction while respecting the database semantics?

We focus on model checking the business and data tiers of web applications, assuming the correctness of the database management system. We have implemented a model checker DPF (for *Database PathFinder*) for Java programs.

A straightforward model checking approach would model the database interactions using reads and writes on shared-memory objects representing the objects implemented in the database and use standard explicit-state model checking [12, 14, 28]. Unfortunately, we show that such an approach does not scale. Database queries have complex semantics, and their correct modeling brings in too many details of the database implementation. Instead, we represent the application as a multi-threaded imperative program interacting with a database through a core SQL-like declarative query language, and precisely model the semantics of a relational database in the semantics of the programming language. That is, the model checker represents database state as a set of relations, and directly models integrity constraints on the data, such as primary key constraints. The actual database is run along with the model checker to store the concrete relations.

We explore several design choices in our model checker. First, we explore stateful vs. stateless search. In stateful search, we implement two approaches for backtracking the database state. In the first approach, we exploit the savepoint and rollback mechanism of the database, so we can roll back the database state at a backtrack point. In the second approach, we replay the queries performed on the database from an initial state to come to a backtrack point.

Second, we explore partial-order reduction strategies at various granularity levels. Partial-order reduction (POR) requires identifying when two operations are dependent. We give conditions to identify dependent operations at the database, relation, attribute, record, and cell levels. Dependencies are more precise as we go down the levels from database to cell, but require more bookkeeping. We experimentally evaluate the effect of POR at different levels: we show that the number of states explored decreases from naïve exhaustive exploration to cell level, with over $20\times$ reduction in some cases.

Our implementation and evaluation on 12 programs suggests that model checking can be an effective tool for ensuring correctness of web applications. In our experiments, we were able to find concurrency errors in two open-source Java-based web applications. Specifically, we found concurrency errors in PetClinic, an example e-commerce application distributed with the Spring framework, and in OpenMRS, an open-source medical records system. In each case, the programmers had not considered conflicting but non-atomic accesses made concurrently to the database.

While much of our model checker specializes general model-checking techniques, our contributions include: (1) design choices that customize the model checker (including stateful/stateless search, state storage, and backtracking) to the domain of database-backed applications (Section 3), (2) domain-specific versions of partial-order reduction (Section 4), and (3) empirical validation that model checking can be quite effective in detecting concurrency errors for database-backed applications (Section 5).

2 Modeling Database Applications as Transition Systems

We formalize our model-checking algorithm for a concurrent imperative language that accesses a relational database using a simplified structured query language (SQL). We model the semantics of the database as in [5], but additionally allow multiple threads of execution in the program interacting with the database.

Relational Databases. A *relational schema* is a finite set of relation symbols with associated arities. Each relation symbol is an ordered list of named *attributes*; an attribute is used to identify each position. A record is an ordered list of attribute values. The value of attribute A has position $\mathsf{pos}(A)$. A finite relation is a finite set of records. For simplicity of exposition, we assume that each attribute value is an integer; our implementation handles all the datatypes supported by a database.

A *relational database* over a relational schema S represents a mapping from relation symbol $\mathbf{r} \in \mathsf{S}$ to finite relation r, such that r has the same arity as $\mathbf{r}$. We write $r \cup \{\rho\}$ to denote an extension of the relation r with the record ρ, which has the same arity as r, and $r \setminus \{\rho\}$ to denote a removal of the record ρ from the relation r. A *key* can be defined on a relation symbol $\mathbf{r}$ to identify an attribute that has a unique value in each record of the relation; $\mathsf{kdef}(\mathbf{r})$ holds if a key is defined on $\mathbf{r}$ and $\mathsf{key}(\mathbf{r})$ returns the position of that key. Finally, for a relational schema S, a relation symbol $\mathbf{r} \in \mathsf{S}$, a finite relation r of the same arity as $\mathbf{r}$, and a database R over S, we write $R[\mathbf{r} \leftarrow r]$ for the relational database where $\mathbf{r}$ is mapped to r, while all other relation symbols are the same as in R.

Structured Query Language (SQL). We assume that the program communicates with the database using a declarative query language. We focus on a simplified data manipulation language that allows querying, insertion, deletion, or update of the relations in the database. The syntax of the language is as follows.

1. The SELECT statement SELECT $A_1, \ldots, A_k$ FROM $\mathbf{r}$ WHERE ψ queries the database and returns a relation with attributes $A_1, \ldots, A_k$, such that for each record r in the relation, there is a record in the relation $\mathbf{r}$ that agrees with r on $A_1, \ldots, A_k$ and also satisfies the predicate ψ.
2. The INSERT statement INSERT INTO $\mathbf{r}$ VALUES $(v_1,...,v_n)$ inserts a new record in the relation $\mathbf{r}$, if the integrity constraints are satisfied.
3. The DELETE statement DELETE FROM $\mathbf{r}$ WHERE ψ removes all records from the relation $\mathbf{r}$ that satisfy the predicate ψ.
4. The UPDATE statement UPDATE $\mathbf{r}$ SET $A = F(A)$ WHERE ψ updates the values of an attribute by applying the function F in all records that satisfy the predicate ψ, if the integrity constraints are satisfied.

Note that insertions and updates check integrity constraints on the database. These are invariants on the database that are specified at the schema level. For example, the PRIMARY KEY constraint requires that each value of a key attribute appears at most once in a relation.

We formalize the semantics of the SQL statements in a straightforward way and additionally include support for integrity constraints. We fix a schema S consisting of a set of relation symbols. A database R over S consists of a set of relations, one for each relation symbol $\mathbf{r}$ in S of the same arity as $\mathbf{r}$ and with the same attributes.

We first define some standard functions. For a relation $r = R(\mathbf{r})$ and predicate ψ over the attributes of $\mathbf{r}$, we define the *selection function* $\sigma_\psi(r)$ as a relation that includes all records that satisfy the condition ψ: $\{\rho \mid \rho \in r \wedge \rho \models \psi\}$. The *projection* $\pi_{A_1,\ldots,A_k}(r)$ projects a relation r to only the attributes $A_1, \ldots, A_k$. The substitution $r[A \leftarrow F(A)]$, for a function F mapping integers to integers is defined as $\{\langle v_1,...,v_{i-1},F(v_i),v_{i+1},...,v_n\rangle \mid \langle v_1,...,v_n\rangle \in r \wedge i = \mathsf{pos}(A)\}$.

Lastly, for an attribute K, we define a predicate $\xi_K(r)$ that holds iff each record of r has a unique value on attribute K, i.e., for all $\rho \in r$, $\langle \rho_{\mathsf{pos}(K)}\rangle \notin \pi_K(r \setminus \rho)$.

The semantics of each SQL statement can now be given as a transformer on the database and an output relation (representing the result of the SQL statement).

Select: For a given set of attributes $\{A_i \mid i \in \{1,...,k\}\}$ of a relation $\mathbf{r}$, the select operation returns a pair of the unmodified relational database and a set of records that satisfy the predicate ψ and projected on $A_1,...,A_k$: $\langle R,\pi_{A_1,...,A_k}(\sigma_\psi(R(\mathbf{r})))\rangle$.

Insert: For a record $\langle v_1,...,v_n\rangle$ of the same arity as $\mathbf{r}$, the insert operation returns a relational database that includes a new record in the mapping for relation $\mathbf{r}$ if a key is not defined on $\mathbf{r}$, or the record has a unique value on the key attribute; otherwise, it returns the original relational database and an empty output relation:

$$\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \cup \{\langle v_1,...,v_n\rangle\}],\emptyset\rangle, \text{ if } \neg\mathsf{kdef}(\mathbf{r}) \ \vee \ \xi_{\mathsf{key}(\mathbf{r})}(R'(\mathbf{r}))$$
$$\langle R,\emptyset\rangle \text{ , otherwise.}$$

Delete: The delete operation creates a new mapping for the relation symbol $\mathbf{r}$, which includes all the records that do not satisfy the predicate ψ and an empty output relation: $\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \setminus \sigma_\psi(R(\mathbf{r}))],\emptyset\rangle$

Update: The update operation creates a new mapping for the relation symbol $\mathbf{r}$, which includes all records that do not satisfy the predicate ψ and all records that satisfy the predicate ψ with substituted value for A. The update is possible if either a key is not defined on $\mathbf{r}$ or the integrity constraints are satisfied after the update; otherwise, the operation does not modify the relational database:

$$\begin{array}{c}\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \setminus \sigma_\psi(R(\mathbf{r})) \cup \sigma_\psi(R(\mathbf{r}))[A \leftarrow F(A)]], \emptyset\rangle, \text{ if } \neg\mathsf{kdef}(\mathbf{r}) \ \vee\ \xi_{\mathsf{key}(\mathbf{r})}(R'(\mathbf{r})) \\ \langle R, \emptyset\rangle \text{ , otherwise.}\end{array}$$

Multithreaded Database Programs. Our program model consists of multithreaded imperative programs with a fixed number of threads, where each thread has its own local variables (including relation-valued variables). In addition to usual assignment, conditional, and looping constructs, each thread can make SQL statements to a shared database with a fixed schema S. We assume w.l.o.g. that there is no global shared memory state (but our implementation supports additional global state as well). In order to model transactions, we assume that a statement can be enclosed within a keyword "transaction," and the database implementation guarantees that the entire transaction executes atomically. We omit a concrete syntax for our programs.

The state of a program consists of the state of the relational database and local states of each thread:

$$\begin{array}{c}\text{State of the program} = \text{State of the database} \times \text{States of the threads} \\ \text{States of the threads} = \text{a map from each thread to its local state} \\ \text{Local state} = \text{Location in the program} \times \text{a map from each local var to its value}\end{array}$$

At the beginning of the execution of a program, each local state is in an initial state, where the program counter is at the starting location for each thread and each value is initialized to the default value of the appropriate type. The initial state of the database is provided by the user and represents the state at the beginning of an execution. Note that the result of the exploration depends crucially on the initial database state.

The execution of a program advances by performing the following steps in a loop: (1) select a thread non-deterministically from the set of live threads, and (2) execute the statements of the thread until the thread finishes the execution or is about to exit from a transaction statement (this represents a commit of the transaction); the sequence of statements executed by a thread without interruption is called a *transition* [8]. The program execution ends when all threads finish the execution.

Note that while two transactions can be interleaved in a concrete run of a program, we rely on the correctness properties of the database implementation to run transactions atomically and in isolation. Moreover, we rely on a lower-level primitive that performs retries in case a transaction must be aborted, so that we only check execution paths of the program in which transactions commit.

Explicit-State Model Checking. Now that we have modeled a database application as a state-transition system, we can implement an explicit-state model checker that systematically explores all interleavings of the program [12, 14, 28]. Since the database is the only shared state, the only *visible* operations of the program (i.e., points at which thread switches must be scheduled) are transaction statements.

In the next two sections, we discuss key implementation choices in the model checker: (1) representation of the database (in-memory vs. on-disk), (2) state storage, checkpointing and restoration, and matching, and (3) partial-order reduction.

3 Design Decisions and Implementation

Extending JPF. We implemented our model checker DPF (Database PathFinder) for Java programs as an extension of Java PathFinder (JPF) [16,28]. JPF is a popular, extensible and configurable model checker for Java programs; a user can select stateful or stateless search, state storing and restoring mechanism, state matching algorithm, search heuristic, etc. JPF implements a backtrackable Java Virtual Machine that explores a program by interpreting bytecode instructions. As a consequence, JPF does not support (i.e., cannot analyze) Java programs that employ *native* methods, since the native methods are not implemented as java bytecode. To overcome the limitations one has to implement native methods to operate on JPF's internal memory representation.

Although in principle JPF should be able to explore any Java program, including database applications, there are significant challenges in applying it in our context. First, JPF is unable to execute database applications because applications use native methods, e.g., from Java Database Connectivity (JDBC) used by Java programs to interact with databases. Second, JPF sees accesses to shared-memory objects as the only source of non-determinism.[1] Therefore, JPF does not consider database transactions as the scheduling points, which are the key scheduling points for database applications. Thus, JPF will not explore their interleavings. Third, JPF may perform incorrect execution of database applications. JPF stores the in-memory state of an application at each scheduling point and restores the state when it explores the next choice from that point. This ignores the state of the database (and of external files). Similarly, JPF stores hashes only of in-memory objects, and will ignore the database state when matching the state in stateful exploration. Next, we describe how we customized JPF to address these challenges and enable the (optimized) exploration of database applications.

Design 1: In-Memory Database. The simplest approach to address all the challenges is to configure a database application to use an in-memory database, e.g., H2[2], which uses data structures in memory to represent a database but exposes a SQL interface to these data structures. Consequently: 1) there are no accesses to external resources (although we still had to implement a few native methods used by H2), 2) the database becomes, from the JPF perspective, an in-memory object shared among threads and therefore JPF schedules all relevant threads at each access to the data structures that represent the database, 3) the database is stored/restored by JPF at scheduling points as any other in-memory object, and 4) state matching hashes the state of the database (at the *concrete* level of the data structures that implement the database rather than at the *abstract* level of the database).

However, this approach has a number of drawbacks. JPF explores the implementation of the database together with an application, therefore introducing unnecessary overhead. We would rather explore the semantics of the application assuming correctness of the DBMS implementation. Next, keeping a database in JPF memory is not acceptable for any realistic application with many records. Also, state matching is not optimal, since many internal structures are part of the state (e.g., different orders of two records lead to different states, even when they encode the same relation).

[1] JPF also supports data non-determinism, but that is irrelevant for our discussion.

[2] http://h2database.com/

Our experimental evaluation showed that this technique does not complete in reasonable time or space for any real application, and not even for micro-benchmarks.

Design 2: On-Disk Database. To enable JPF to work with on-disk databases, we first intercept the native methods from the JDBC API and extended JPF to view some of these methods as scheduling points. We next extend JPF to restore the database, and so support stateless exploration even when the application updates the database. Our first approach to tackle this problem was to intercept all method invocations that access the database and save the SQL operations used in these accesses. We keep a mapping from each memory state that JPF encounters during exploration to the sequence of SQL operations executed up to that state. When JPF restores the memory state, we additionally first restore the database to what it was at the beginning of the exploration and then replay the saved SQL operations that correspond to the state being restored. While this approach correctly restores the database for each state, we recognized that in some cases we may further optimize state restoring by leveraging the roll back mechanism of databases. In our second approach, for each new state in JPF, we set a *savepoint* [3]. Then, when JPF restores the memory state, we instruct the database to roll back to the appropriate savepoint. Note that the second approach works only for depth-first search exploration because there can be at most one sequence of savepoints in a database.

To further optimize the exploration, we consider stateful search. Before each state matching, we compute the hash of the database and add it to the hash that JPF computes for the memory state. We explore two ways of hashing databases: the *full* approach that computes hash of the entire database each time, and the *incremental* approach that updates the hash value each time the database is changed. Note that the incremental hash function must be commutative [15, 20]; otherwise, the same set of records may lead to different hash values if they are inserted in different order. Our current implementation modifies the H2 database to support incremental hashing.

4 Partial-Order Reduction

POR [8, 11, 12, 30] is an optimization technique that exploits the fact that many paths are redundant as they execute independent transitions in different orders. Two transitions are *independent* [11] if their executions do not affect each other, i.e., if the two transitions commute and do not disable each other. The *naïve* approach (i.e., without POR) trivially considers any two operations to be dependent and exhaustively explores the entire transition graph. POR techniques identify dependent transitions and explore a set of paths that is a subset of the paths that are executed by the naïve approach.

For database applications, we can track dependencies among SQL operations at different granularity levels: *database*, *relation*, *attribute*, *record*, and *cell*. These levels differ in precision of the tracked information (thus enabling more pruning in the exploration) and in the cost of tracking that information (the more precise ones are more expensive to track). We now describe the dependency conditions for these levels.

Figure 1 compactly presents the sufficient conditions to identify dependent transitions for more precise granularities. These conditions assume that there is one SQL operation per transition and that both transitions use the same relation symbol $\mathbf{r}$ (as

Transitions	Granularity			
	Relation	Attribute	Record	Cell
$\langle \mathsf{S}^1,\mathsf{S}^2\rangle$	$false$	$false$	$false$	Attribute $\wedge$ Record
$\langle \mathsf{S}^1,\mathsf{I}^2\rangle$	$true$	$true$	$\sigma_{\psi^1}(\{\langle v_1^2,\ldots,v_n^2\rangle\}) \neq \emptyset$	
$\langle \mathsf{S}^1,\mathsf{D}^2\rangle$	$true$	$true$	$\sigma_{\psi^1}(R(\mathbf{r})) \cap \sigma_{\psi^2}(R(\mathbf{r})) \neq \emptyset$	
$\langle \mathsf{S}^1,\mathsf{U}^2\rangle$	$true$	$A^2 \in \{A_1^1,\ldots,A_k^1\}$	$\sigma_{\psi^1}(R(\mathbf{r})) \neq \sigma_{\psi^1}(R^{\mathsf{U}^2}(\mathbf{r}))$	
$\langle \mathsf{I}^1,\mathsf{I}^2\rangle$	$true$	$true$	$false \curlyvee \pi_K(\{\langle v_1^1,\ldots,v_n^1\rangle\}) \cap \pi_K(\{\langle v_1^2,\ldots,v_n^2\rangle\}) \neq \emptyset$	
$\langle \mathsf{I}^1,\mathsf{D}^2\rangle$	$true$	$true$	$\sigma_{\psi^2}(\{\langle v_1^1,\ldots,v_n^1\rangle\}) \neq \emptyset$	
$\langle \mathsf{I}^1,\mathsf{U}^2\rangle$	$true$	$true$	$\sigma_{\psi^2}(\{\langle v_1^1,\ldots,v_n^1\rangle\}) \neq \emptyset \curlyvee \pi_K(\{\langle v_1^1,\ldots,v_n^1\rangle\}) \cap \pi_K(R^{\mathsf{U}^2}(\mathbf{r})) \neq \emptyset$	
$\langle \mathsf{D}^1,\mathsf{D}^2\rangle$	$false$	$false$	$false$	
$\langle \mathsf{D}^1,\mathsf{U}^2\rangle$	$true$	$true$	$\sigma_{\psi^1}(R(\mathbf{r})) \neq \sigma_{\psi^1}(R^{\mathsf{U}^2}(\mathbf{r})) \curlyvee \sigma_{\psi^2}(R(\mathbf{r})) \neq \sigma_{\psi^2}(R^{\mathsf{D}^1}(\mathbf{r}))$	
$\langle \mathsf{U}^1,\mathsf{U}^2\rangle$	$true$	$A^1 = A^2$	$\sigma_{\psi^1}(R(\mathbf{r})) \neq \sigma_{\psi^1}(R^{\mathsf{U}^2}(\mathbf{r})) \vee \sigma_{\psi^2}(R(\mathbf{r})) \neq \sigma_{\psi^2}(R^{\mathsf{U}^1}(\mathbf{r})) \curlyvee$ $(\pi_K(R^{\mathsf{U}^1}(\mathbf{r})) \setminus \pi_K(R(\mathbf{r}))) \cap (\pi_K(R^{\mathsf{U}^2}(\mathbf{r})) \setminus \pi_K(R(\mathbf{r}))) \neq \emptyset$	

Fig. 1. Conditions to identify dependent transitions where $\mathbf{r}^1 = \mathbf{r}^2$

transitions on different relations are trivially independent in our language). Recall that $\mathsf{kdef}(\mathbf{r})$ holds if the relation symbol has a key; if so, we assume the key is called K.

Figure 1 uses the following notation. A pair of transitions is named by the first letter of their SQL operations, e.g., $\langle \mathsf{S}^1,\mathsf{I}^2\rangle$ stands for $\langle \mathsf{SELECT}^1,\mathsf{INSERT}^2\rangle$. Recall that these operations have parameters; we use X^1 and X^2 to refer to the parameters of the operations, e.g., in an expression $\sigma_{\psi^1}(\{\langle v_1^2,\ldots,v_n^2\rangle\})$, ψ^1 is the predicate used in the first transition and $v_1^2,\ldots,v_n^2$ are the attributes used in the second transition. $R^{\mathcal{O}}$ refers to the database after the operation $\mathcal{O}$ is executed. Finally, symbol $\curlyvee$ splits a condition in the part sufficient if a primary key is not defined on the relation and the part that is additionally sufficient if a primary key is defined; other than that, $\curlyvee$ is equivalent to logical $\vee$ operator; the conditions are weaker for relations that have keys because the implicit constraints preclude some insert/update operations.

Relation Granularity. Most pairs of transitions are (conservatively) marked as dependent. For example, consider $\langle \mathsf{S}^1,\mathsf{I}^2\rangle$ (the second row in Figure 1), which are dependent if the record to be inserted by I^2 would be selected by S^1. Because the only available information at the relation granularity level is the name of the relation used in the operations, it is not possible to know if S^1 would select the record inserted by I^2. However, the relation granularity is still more precise than the naïve exploration for the two cases ($\langle \mathsf{S}^1,\mathsf{S}^2\rangle$ and $\langle \mathsf{D}^1,\mathsf{D}^2\rangle$) when the transitions are always independent. First, a SELECT operation does not modify the state of the database, so two SELECT operations are independent (this is similar to read-read independence in shared-memory programs). Second, two DELETE operations are independent, because a DELETE operation either removes all the records that satisfy the predicate or does not affect the database if no record satisfies the predicate (Section 2). Thus, two DELETE operations commute, and the set of removed records is the union of the sets of records removed by these operations. Note that we do not consider all options of databases (e.g., foreign keys, observing failing operations, etc.), which may change the notion of dependence in some cases. For example, two DELETE statements may not commute if a foreign key is defined because the first delete may remove the records such that the second delete cannot execute. Also, while database and relation granularities are the same for our language when $\mathbf{r}^1 = \mathbf{r}^2$, these granularities may differ when other options of databases are considered.

Attribute Granularity. The attribute granularity is more precise than the relation granularity, i.e., the transitions that are dependent according to the attribute granularity are dependent according to the relation granularity, while the opposite may not hold. At the attribute granularity level, the extracted information includes a set of attributes used by each transition. Consider two rows in Figure 1 ($\langle \mathsf{S}^1, \mathsf{U}^2 \rangle$ and $\langle \mathsf{U}^1, \mathsf{U}^2 \rangle$) when the attribute granularity may give more precise result. S and U are dependent if the attribute whose values are to be updated is in the set of attributes to be selected. Similarly, U and U are dependent if both update the same attribute; note that even when the attribute is the same, the transitions may actually be independent if they modify different records.

Record Granularity. The record granularity is never less precise than the relation granularity but is incomparable to the attribute granularity. The conditions that are sufficient for two transitions to be dependent are as follows. For $\langle \mathsf{S}^1, \mathsf{I}^2 \rangle$, it suffices to check if the record to be inserted by I^2 would be selected by S^1. $\langle \mathsf{S}^1, \mathsf{D}^2 \rangle$ requires that at least one of the records to be deleted by D^2 be in the set of records to be selected by S^1. $\langle \mathsf{S}^1, \mathsf{U}^2 \rangle$ requires that the sets of records selected by S^1 before and after the update differ. If a key is not defined, $\langle \mathsf{I}^1, \mathsf{I}^2 \rangle$ are never dependent because both records can be inserted in the database. However, if a key is defined, these transitions are dependent if the values of the keys to be inserted are the same. $\langle \mathsf{I}^1, \mathsf{D}^2 \rangle$ requires that the record to be inserted by I^1 would be deleted by D^2. If a key is not defined, $\langle \mathsf{I}^1, \mathsf{U}^2 \rangle$ are dependent if the record to be inserted would be updated; if a key is defined, the transitions are dependent if the record to be inserted and any updated record have the same key value. $\langle \mathsf{D}^1, \mathsf{U}^2 \rangle$ requires different set of records to be selected by D^1 before and after the update if a key is not defined. If a key is defined, the transitions are dependent if a set of records to be selected by U^2 before and after delete is different. For $\langle \mathsf{U}^1, \mathsf{U}^2 \rangle$, if a key is not defined, different sets of records should be selected using the condition of one operation on the database before and after the other operation is executed. If a key is defined, the transitions are dependent if they would insert any records that have the same key value.

Cell Granularity. The cell granularity combines the power of attribute and record granularities to identify values in the records that are accessed by each operation. This makes the cell granularity the most precise. The conditions are conjunction of conditions required for attribute and record granularities.

We implemented dependency analysis for relation and cell granularities in DPF. Since the set of relation symbols used in the SQL statements does not depend on the database content, our implementation caches the set of relation symbols for each operation and uses the cache in dependency analysis for the relation granularity.

POR vs. Database Management System (DBMS) Serializability. DBMS checks if two transitions are serializable when they execute concurrently [3]. In contrast, DPF checks if two transactions *commute* even if they are executed serially. DBMS employs mechanisms, such as read/write sets [3], to answer the serializability question. While it may be possible to use these mechanisms to check dependence, our straightforward implementation of dependence-tracking using DBMS conflict detection was imprecise as the read/write sets involved implementation-specific objects such as locks that may not affect dependence. Thus, we implemented the semantic notions described above.

5 Experimental Evaluation

We now present a performance evaluation of DPF and describe three bugs we found.

Configurations. First, we evaluate DPF with in-memory database. Second, we evaluate DPF with on-disk database combined with one of two approaches to restore the database (Replay or Rollback), one of two approaches to hash the database (Full or Incremental), and one of three POR granularity levels (Naïve, Relation, or Cell). Therefore, we have 13 configurations of DPF. All experiments were performed on a machine with a 4-core Intel Core i7 2.70GHz processor and 4GB of main memory, running Linux version 3.2.0, and Java Oracle 64-Bit Server VM, version 1.6.0_33.

Benchmarks. Our set of benchmarks includes four real-world applications and 7 kernels that we used to evaluate DPF. The applications are as follows: `OpenMRS`[3] is an open-source enterprise electronic medical record system platform with 150K lines of Java code in the core repository; `PetClinic`[4] is an official sample distributed with the Spring framework [27] and implements an information system to be used by a veterinary clinic to manage information about veterinarians, pet owners, and pets; `RiskIt`[5] is an insurance quote application with 13 relations, 57 attributes, and more than 1.2 million records; and `UCOM`[6] is a program for obtaining statistics about usage of a system. `RiskIt` and `UCOM` have been used in previous research studies on database applications [13,23,24]. The kernels include these: `InsertDelete` is created to test the $\langle \mathsf{I},\mathsf{D} \rangle$ dependency; `IndAtts` is created to test POR with the attribute granularity by spawning a couple of threads that update values of different attributes; `IndCells` is created to test POR with the cell granularity by spawning multiple threads that use different cells; `IndD` is created to test the dependency among delete operations; `IndRels` is created to test POR with the relation granularity; `Accesses` spawns threads that perform many (independent and dependent) SQL operations; and `Entries` is created to test our two approaches for hashing the database.

Tests. As for other dynamic techniques, DPF requires an input that initiates the exploration. While our kernels do not require any input, we had to construct test inputs for four applications. Unfortunately, none of the applications include concurrent test cases. (A concurrent test case spawns two or more threads.) However, all applications include a (large) number of sequential test cases. We created concurrent test cases by combining the existing sequential test cases; each sequential test case is executed by one thread. Combining sequential test cases can be challenging and currently we mostly do it manually. In the future, we would like to investigate in more detail the power of concurrent test cases that are obtained by combining sequential test cases. Also, we would like

[3] `http://openmrs.org` (version 1.9.1)
[4] `http://static.springsource.org/docs/petclinic.html` (revision 616)
[5] `https://riskitinsurance.svn.sourceforge.net` (revision 96)
[6] `http://sourceforge.net/projects/redactapps` (version of October 14, 2012)

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.
▼ Benchmark	DB Type ▶	In-memory	On-disk											
	DB Hashing ▶	na	Full						Incremental					
	DB Restore ▶	na	Replay			Rollback			Replay			Rollback		
	Granularity ▶	na	Naïve	Rel.	Cell	Naïve	Rel.	Cell	Naïve	Rel.	Cell	Naïve	Rel.	Cell
	Statistics ▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼
Applications														
OpenMRS	Expl [ms]	-	53513	3201	3082	145881	3121	3137	53482	2634	3095	149579	3098	3109
	Speedup [X]	na	na	16.72	17.36	na	46.74	46.50	na	20.30	17.28	na	48.28	48.11
	#States	-	33855	2193	1625	33855	2193	1625	33855	2193	1625	33855	2193	1625
	Reduction [X]	na	na	15.44	20.83	na	15.44	20.83	na	15.44	20.83	na	15.44	20.83
	#Trans	-	68182	3221	2214	68182	3221	2214	68182	3221	2214	68182	3221	2214
	Memory [MB]	-	437	102	104	460	106	104	437	102	102	442	102	104
	Hash [ms]	na	89	15	9	100	10	16	17	2	4	27	1	1
PetClinic1	Expl [ms]	-	107225	20334	14223	94448	18188	13343	106107	20976	14682	93750	18714	14056
	Speedup [X]	na	na	5.27	7.54	na	5.19	7.08	na	5.06	7.23	na	5.01	6.67
	#States	-	39571	10919	5646	39571	10919	5646	39571	10919	5646	39571	10919	5646
	Reduction [X]	na	na	3.62	7.01	na	3.62	7.01	na	3.62	7.01	na	3.62	7.01
	#Trans	-	52733	13962	6708	52733	13962	6708	52733	13962	6708	52733	13962	6708
	Memory [MB]	-	610	503	387	541	515	393	617	499	393	577	509	390
	Hash [ms]	na	295	73	56	305	84	57	27	16	37	31	11	36
PetClinic2	Expl [ms]	-	4549	3359	3589	3709	2785	3005	4647	3495	3657	3599	2822	2976
	Speedup [X]	na	na	1.35	1.27	na	1.33	1.23	na	1.33	1.27	na	1.28	1.21
	#States	-	1234	458	451	1234	458	451	1234	458	451	1234	458	451
	Reduction [X]	na	na	2.69	2.74	na	2.69	2.74	na	2.69	2.74	na	2.69	2.74
	#Trans	-	1744	620	609	1744	620	609	1744	620	609	1744	620	609
	Memory [MB]	-	167	102	102	102	102	105	101	102	106	106	102	102
	Hash [ms]	na	19	12	10	11	16	9	4	2	2	2	0	3
RiskIt	Expl [ms]	-	1012343	615514	391515	1050192	632422	397537	26447	25969	25805	26509	25982	26635
	Speedup [X]	na	na	1.64	2.59	na	1.66	2.64	na	1.02	1.02	na	1.02	1.00
	#States	-	706	276	203	706	276	203	706	276	203	706	276	203
	Reduction [X]	na	na	2.56	3.48	na	2.56	3.48	na	2.56	3.48	na	2.56	3.48
	#Trans	-	1349	415	278	1349	415	278	1349	415	278	1349	415	278
	Memory [MB]	-	368	368	367	367	368	366	368	368	368	368	366	368
	Hash [ms]	na	167227	95498	58708	168815	100173	62133	7	2	5	9	5	7
UCOM	Expl [ms]	-	173510	39289	39262	177845	41901	41631	12102	9439	9545	11999	9426	9883
	Speedup [X]	na	na	4.42	4.42	na	4.24	4.27	na	1.28	1.27	na	1.27	1.21
	#States	-	1152	433	416	1152	433	416	1152	433	416	1152	433	416
	Reduction [X]	na	na	2.66	2.77	na	2.66	2.77	na	2.66	2.77	na	2.66	2.77
	#Trans	-	3421	822	775	3421	822	775	3421	822	775	3421	822	775
	Memory [MB]	-	371	370	370	370	370	370	371	365	371	371	365	371
	Hash [ms]	na	35141	6664	6705	36121	7320	7153	8	0	2	14	2	5
Kernels														
InsertDelete	Expl [ms]	4427	1078	1072	1075	1085	1078	1102	1097	1079	1080	1050	1061	1081
	#States	898	40	36	36	40	36	36	40	36	36	40	36	36
	#Trans	1416	66	54	54	66	54	54	66	54	54	66	54	54
	Memory [MB]	177	72	72	72	72	72	72	57	72	72	57	57	57
	Hash [ms]	na	1	0	2	1	3	4	0	0	1	0	2	0
IndAtts	Expl [ms]	723195	4742	4344	5219	5482	5273	5821	4665	4755	5320	5523	5215	6003
	#States	168561	3269	3153	3057	3269	3153	3057	3269	3153	3057	3269	3153	3057
	#Trans	361251	9975	8561	8369	9975	8561	8369	9975	8561	8369	9975	8561	8369
	Memory [MB]	428	165	284	141	165	236	165	165	155	141	236	198	229
	Hash [ms]	na	50	57	72	43	58	71	8	15	28	10	23	27
IndCells	Expl [ms]	115328	2314	1811	2512	2368	2244	2497	2325	2292	2558	2391	2218	2562
	#States	26693	1061	981	921	1061	981	921	1061	981	921	1061	981	921
	#Trans	64472	3055	2405	2285	3055	2405	2285	3055	2405	2285	3055	2405	2285
	Memory [MB]	249	102	102	105	102	99	102	104	99	102	105	99	104
	Hash [ms]	na	11	10	17	12	6	7	1	1	11	3	2	21
IndD	Expl [ms]	-	19133	12693	13174	45553	16009	17970	18835	12414	13117	44644	16051	17381
	#States	-	15610	11732	11732	15610	11732	11732	15610	11732	11732	15610	11732	11732
	#Trans	-	58277	35005	35005	58277	35005	35005	58277	35005	35005	58277	35005	35005
	Memory [MB]	-	294	352	372	284	369	390	202	285	284	284	327	298
	Hash [ms]	na	38	42	34	44	46	51	17	17	26	22	34	26
IndRels	Expl [ms]	113317	1902	2326	2499	2452	2287	2382	2375	2219	2418	2438	2176	2332
	#States	26693	1061	921	921	1061	921	921	1061	921	921	1061	921	921
	#Trans	64472	3055	2285	2285	3055	2285	2285	3055	2285	2285	3055	2285	2285
	Memory [MB]	254	102	102	102	102	105	104	102	104	104	104	105	105
	Hash [ms]	na	7	13	6	14	15	7	0	3	9	3	5	8
Accesses	Expl [ms]	-	18396	10231	3170	14363	8689	3069	18365	10860	3219	14376	8347	3084
	#States	-	14856	8759	369	14856	8759	369	14856	8759	369	14856	8759	369
	#Trans	-	26125	11884	590	26125	11884	590	26125	11884	590	26125	11884	590
	Memory [MB]	-	154	327	84	214	286	72	194	381	72	139	223	65
	Hash [ms]	na	93	68	26	102	57	19	28	20	29	18	18	16
Entries	Expl [ms]	-	4675	3751	3190	3442	2900	3014	4215	3218	2979	2856	2495	2628
	#States	-	467	302	139	467	302	139	467	302	139	467	302	139
	#Trans	-	819	426	215	819	426	215	819	426	215	819	426	215
	Memory [MB]	-	102	105	102	155	155	105	105	72	74	104	72	72
	Hash [ms]	na	400	324	223	405	314	252	2	2	5	4	3	6

Fig. 2. Exploration statistics for multiple DPF configurations

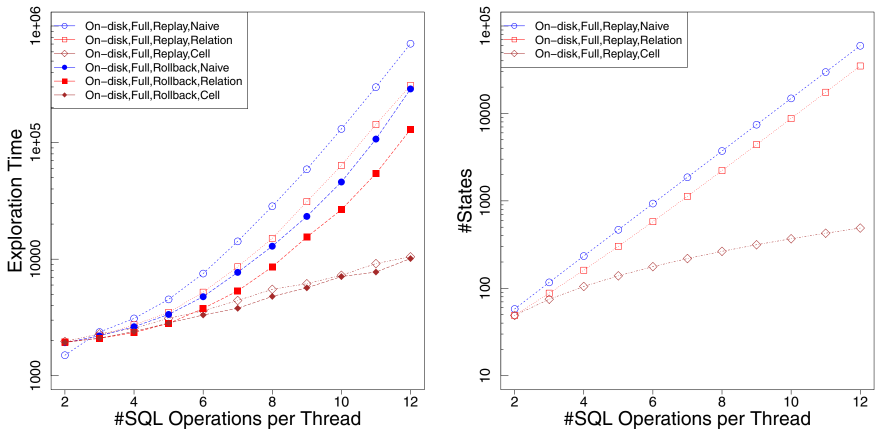


Fig. 3. Exploration time (left) and number of states (right) for the `Entries` benchmark

to combine DPF with the existing approaches [6, 18] to discover entry points in web applications and run multiple threads that use these points.

Results. Figure 2 shows, for each benchmark (listed in column 1) several statistics (column 2) for each of the 13 evaluated configurations of DPF (columns 3–15). Specifically, we show the exploration time, speedup in time (over Naïve approach), number of explored states, reduction in the state space (over Naïve approach), number of transitions, memory usage, and time for hashing the database; because of space limit we show speedup and reduction only for the applications.

We can observe the following. (1) In-memory database is not acceptable even for small examples, e.g., comparing columns 3 and 4 for `IndCells`, the exploration time is over 50x slower for in-memory database. In fact, using in-memory database, DPF often runs out of memory or time limit (set to 1h), marked with "-". (2) More precise POR granularity can significantly reduce the exploration time, e.g., looking at columns 4 to 6 for `Accesses`, going from Naïve to Relation reduced the time 2x and going from Relation to Cell reduces the time 3x more. However, more precise POR granularity does not always result in smaller exploration time because more precise POR granularity has additional cost to compute the dependency more precisely, e.g., for `IndCells` columns 5 and 6, the time for Relation is smaller than the time for Cell. Therefore, less precise POR could perform better when the number of explored states is small and there are no independent accesses, which is almost never the case for real applications [12]. Additionally, if there is a task that should be performed at each state, more precise granularity yields significantly better results even for small number of states, e.g., for `RiskIt`, columns 4 and 6 show significant improvement compared to columns 13 to 15. (3) Using Rollback to restore the database is not always faster than Replay, e.g., for `OpenMRS` columns 4 and 7, the time for Replay is smaller than the time for Rollback. We noticed that Rollback does well if the state space graph is closer to being a tree (i.e., the ratio of number of states and transitions is closer to 1). (4) Incremental always takes less time than Full for the hashing process itself, and when hashing time becomes a substantial part of exploration, Incremental also significantly reduces the exploration time, e.g., compare `RiskIt` columns 4 and 10.

Scalability. Figure 3 illustrates scalability of the selected DPF configurations. We show the plots only for `Entries`, because of the space limit, where we parametrized the code to have an increasing number of SQL operations per thread. The plots depict the exploration time (left) and the number of explored states (right) for different number of SQL operations in each thread. (Note that the y axis is in logarithmic scale.) It can be seen that more precise granularity level scales better.

Bugs. While performing the experiments, DPF discovered one bug in `OpenMRS` and two bugs in `PetClinic`. We reported two bugs to the developers [4]. The bugs manifest as uncaught exceptions with a specific schedule of database transactions. The exception in `OpenMRS` happens if two users access the same concept class (e.g., Test, Drug, etc.) and one of the users edits and saves (or deletes) the concept after the second user has already deleted the concept. A bug in `PetClinic` is similar: the exception happens when two users attempt to delete the same pet of the same owner simultaneously. The following schedule leads to the bug: both users access the same owner then the same pet of the owner, and then one sends delete request after delete request by another user has been completed. The second user to send the delete request will get an exception.

We have found a second bug in `PetClinic` when it is configured to use database access through Hibernate. An exception happens if two users access the same owner then the same pet (the execution can be the same as in the bug that we have already reported), and then send delete requests simultaneously. In the delete handler the object is first taken from the database (SQL select) and then deleted (SQL delete), however these two operations are non-atomic and if both users first get the object and then try to delete only the first delete succeeds while the second throws an exception. This bug differs from the first schedule described above. In the first case, the bug occurs when two delete requests on the same object are performed sequentially. In the second, the bug occurs when there is a context switch point after the select of one request when the second request runs to completion, and then the first request fails to delete.

6 Related Work

Detecting Concurrency Issues Related to External Resources. Paleari et al. [22] proposed a dynamic approach to detect dataraces in web applications that interact with databases. The approach analyzes a log file of a single run and identifies dependencies among SQL queries based on the set of relations and attributes that are read/written. The solution ignores program semantics, thus leading to false alarms. Our dependency analysis is more precise and it is used to optimize model checking of database applications without false alarms. Closely related work by Zheng and Zhang [31] applies static analysis to detect atomicity violations in external resources, such as files and databases, in application servers. The difference between their work and ours is the usual distinction between static program analysis and model checking: static analysis can be less precise (i.e., have false positives); but model checking requires setting up the environment to uncover bugs. Since web application code often contains complex language features such as reflection, building up of queries using string operations, etc., static analysis is likely to be imprecise in this domain.

Test Generation for Web Applications. Symbolic techniques have been used in generating test cases for database-backed applications [5,17,24] and in detecting vulnerabilities [6,29]. In contrast to these papers, which focus on data non-determinism for a single thread of the application server, we focus on concurrent interactions of multiple server threads with the database. We assume correctness of DBMS implementation; orthogonal research looks for bugs in databases and transaction models [9,19].

Model-Checking Tools. Explicit-state software model checking [2,12,14,21,28] has been shown useful for finding concurrency bugs. Our contribution is to apply the techniques to the important domain of web applications, and to adapt shared-memory model-checking techniques to checking database interactions. QED [18] is a model checker for web applications that systematically explores sequences of requests to a web application and looks for taint-based vulnerabilities but does not interleave transactions within a request. Artzi et al. [1] described an explicit-state model checker for web applications that does not consider concurrent requests. Petrov et al. [25] developed a tool for detecting data races in client-side web applications.

7 Conclusions and Future Work

DPF is the first step toward scalable systematic exploration tools for database-backed web applications, and much work remains. DPF can be extended to support: (1) other database constraints, e.g., foreign key, that can semantically affect even relations that do not syntactically appear in a SQL operation, (2) operations with multiple relations, e.g., join clause, (3) transactions with multiple SQL operations, and (4) exploration of transactions that can be aborted. In addition, dynamic exploration of closed programs in DPF can be combined with static techniques [31], data non-determinism [5,24], and automatic generation of environment models [10].

Acknowledgments. We thank Don Batory, Yegor Derevenets, Sarfraz Khurshid, Hank Korth, Darko Marinov, Roland Meyer, Francesco Sorrentino, and Samira Tasharofi for discussions about this work; Mark Grechanik, Kai Pan, Kunal Taneja, and Tao Xie for information about benchmarks; and Aleksandar Milicevic for comments on the text. This material is based upon work partially supported by the US NSF under Grant Nos. CCF-1012759 and CCF-0746856. The first author was an intern at the Max Planck Institute for Software Systems during Summer 2012.

References

1. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Softw. Eng. 36(4), 474–494 (2010)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008)
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)

4. DPF home page, http://mir.cs.illinois.edu/~gliga/projects/dpf/
5. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: ISSTA, pp. 151–162 (2007)
6. Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G.: Toward automated detection of logic vulnerabilities in web applications. In: USENIX Security Symposium, pp. 143–160 (2010)
7. Flanagan, C., Freund, S.: Atomizer: A dynamic atomicity checker for multithreaded programs. Sci. Comput. Program. 71(2), 89–109 (2008)
8. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121 (2005)
9. Fonseca, P., Li, C., Rodrigues, R.: Finding complex concurrency bugs in large multi-threaded applications. In: EuroSys, pp. 215–228 (2011)
10. Giannakopoulou, D., Păsăreanu, C.S.: Interface Generation and Compositional Verification in JavaPathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
11. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
12. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL, pp. 174–186 (1997)
13. Grechanik, M., Csallner, C., Fu, C., Xie, Q.: Is data privacy always good for software testing? In: ISSRE, pp. 368–377 (2010)
14. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
15. Iosif, R.: Exploiting heap symmetries in explicit-state model checking of software. In: ASE, pp. 254–261 (2001)
16. JPF home page, http://babelfish.arc.nasa.gov/trac/jpf/
17. Khalek, S.A., Khurshid, S.: Systematic testing of database engines using a relational constraint solver. In: ICST, pp. 50–59 (2011)
18. Martin, M., Lam, M.S.: Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In: USENIX Security Symposium, pp. 31–43 (2008)
19. Mentis, A., Katsaros, P.: Model checking and code generation for transaction processing software. Concurr. Comput.: Pract. Exper. 24(7), 711–722 (2012)
20. Musuvathi, M., Dill, D.L.: An Incremental Heap Canonicalization Algorithm. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 28–42. Springer, Heidelberg (2005)
21. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
22. Paleari, R., Marrone, D., Bruschi, D., Monga, M.: On Race Vulnerabilities in Web Applications. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 126–142. Springer, Heidelberg (2008)
23. Pan, K., Wu, X., Xie, T.: Database state generation via dynamic symbolic execution for coverage criteria. In: DBTest, pp. 1–6 (2011)
24. Pan, K., Wu, X., Xie, T.: Generating program inputs for database application testing. In: ASE, pp. 73–82 (2011)
25. Petrov, B., Vechev, M., Sridharan, M., Dolby, J.: Race detection for web applications. In: PLDI, pp. 251–262 (2012)
26. 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)
27. Spring Framework home page, http://springsource.org/
28. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)

29. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: ISSTA, pp. 249–260 (2008)
30. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient Stateful Dynamic Partial Order Reduction. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 288–305. Springer, Heidelberg (2008)
31. Zheng, Y., Zhang, X.: Static detection of resource contention problems in server-side scripts. In: ICSE, pp. 584–594 (2012)

Efficient Property Preservation Checking of Model Refinements

Anton Wijs and Luc Engelen

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{A.J.Wijs,L.J.P.Engelen}@tue.nl

Abstract. In model-driven software development, models and model refinements are used to create software. To automatically generate correct software from abstract models by means of model refinement, desirable properties of the initial models must be preserved. We propose an explicit-state model checking technique to determine whether refinements are property preserving. We use networks of labelled transition systems (LTSs) to represent models with concurrent components, and formalise refinements as systems of LTS transformation rules. Property preservation checking involves determining how a rule system relates to an input network, and checking bisimilarity between behaviour subjected to transformation and the corresponding behaviour after transformation. In this way, one avoids generating the entire LTS of the new model. Experimental results demonstrate speedups of several orders of magnitude.

1 Introduction

Model-driven software development [2] entails creating implementations on a low level of abstraction from designs represented by models on a high level of abstraction. Implementation details, for example motivated by hardware restrictions, are added incrementally to these abstract models by means of refining model transformations. Usually, an implementation must satisfy a number of requirements that can be expressed as properties of the model that forms its design. Then, the transformations should preserve these properties. Model checking [4] can help to determine whether this is the case, but verifying the properties from scratch for each new model along the development chain not only requires much time, but it is also likely to become unfeasible very quickly, as the related state space of a model tends to grow exponentially when applying a refinement.

In this paper, we present an explicit-state model checking technique tailored for incremental refinement of models of concurrent systems. If the model that forms the initial design of such a system is relatively small, then at this stage, properties can still be verified using traditional techniques based on explicit state space exploration. When a refinement needs to be applied, then instead of the refined model, the technique analyses the formal semantics of the *refinement*, and determines whether application of the refinement is guaranteed to preserve a particular property. This can be either a safety, liveness, or fairness property. This *property-preservation checking* is purely done by reasoning about the structures of Labelled Transition Systems (LTSs), which we use to express the semantics of both the models and the refinements. For a model, the semantics of each process in the model is expressed as an LTS, and the semantics of the complete concurrent system is expressed implicitly by a *network* of LTSs [19], describing how these process LTSs interact. A refinement is formalised as a system of *LTS transformation rules*, where each rule has a left LTS pattern, describing what should be changed, and a right LTS pattern, describing what the result of the change should be. Furthermore, such a system can add to the interaction mechanism of the processes. By focussing on LTSs, our technique is applicable to any modelling language, either to describe models or refinements, whose semantics can be expressed by LTSs.

Mateescu and Wijs [21] developed an automatic technique called *maximal hiding*, which works for a particular, but still very expressive, fragment of the modal μ-calculus [17]. It identifies all system behaviour not relevant for a given property, and hides all this behaviour, i.e. renames the transition labels to τ. Furthermore, it is compatible with *divergence-sensitive branching bisimilarity* (DSBB) [11]. DSBB is a useful equivalence that respects branching-time and cycles of internal behaviour, and is therefore not only suitable for safety properties, but also liveness and fairness properties. The compatibility lies in the fact that if two LTSs are maximally hidden w.r.t. the same property, and they are DSBB, then they both do or do not satisfy the property. By identifying *all* irrelevant behaviour, maximal hiding maximises the potential for a positive DSBB comparison result.

We use these results to focus on the following question: given a model $\mathcal{M}$ satisfying a property φ written in the μ-calculus fragment, and given a system of transformation rules Σ, when and how can we determine whether it is guaranteed that Σ will not structurally alter the maximally hidden LTS of $\mathcal{M}$ when it is applied to it, i.e. will the resulting LTS of the new model $\mathcal{M}'$ be DSBB to the one of $\mathcal{M}$, if both are maximally hidden? As it turns out, this can be done without investigating the LTS of $\mathcal{M}'$ if some reasonable conditions regarding the applicability of Σ on $\mathcal{M}$ are met.

Shifting the focus from models to model refinements implicitly assumes that a modeller likewise focusses on defining refinements to move her initial model to increasingly lower levels of abstraction, and it is in these refinements where she can influence the development. One could also imagine building a dictionary of reusable refinement patterns, including a pattern to, for example, add functionality to cope with lossy communication channels. In our experimental section, we demonstrate that our technique is applicable for such refinements, in fact it runs several orders of magnitude faster than verifying the refined model.

This paper is structured as follows. Section 2 introduces the preliminaries. In Section 3, we formalise LTS transformation. Next, in Section 4, we discuss our technique for determining whether transformations preserve properties. Experimental results are given in Section 5. Section 6 discusses related work, and Section 7 contains conclusions and pointers to future work.

2 Preliminaries

Labelled transition system. An LTS $\mathcal{G}$ is a tuple $\langle \mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G} \rangle$, where $\mathcal{S}_\mathcal{G}$ is a (finite) set of states, $\mathcal{A}_\mathcal{G}$ is a set of actions (including the invisible action τ), $\mathcal{T}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G} \times \mathcal{A}_\mathcal{G} \times \mathcal{S}_\mathcal{G}$ is a transition relation, and $\mathcal{I}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G}$ is a set of initial states. Actions in $\mathcal{A}_\mathcal{G}$ are denoted by a, b, c, etc. We use $s_1 \xrightarrow{a}_\mathcal{G} s_2$ as a shorthand for $\langle s_1, a, s_2 \rangle \in \mathcal{T}_\mathcal{G}$. If $s_1 \xrightarrow{a}_\mathcal{G} s_2$, this means that in $\mathcal{G}$, an action a can be performed in state s_1, leading to state s_2.

Network of LTSs. We represent models consisting of a finite number of finite-state concurrent processes by a number of LTSs and a set of *synchronisation rules* defining how these LTSs interact. For this, we use the concept of *networks of LTSs* [19]. Given an integer $n > 0$, $1..n$ is the set of integers ranging from 1 to n. A vector $\overline{v}$ of size n contains n elements indexed by $1..n$. For $i \in 1..n$, $\overline{v}[i]$ denotes element i in $\overline{v}$.

Definition 1. *A network of LTSs $\mathcal{M}$ of size n is a pair $\langle \Pi, \mathcal{V} \rangle$, where*

- *Π is a vector of n (process) LTSs. For each $i \in 1..n$, we write $\Pi[i] = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{I}_i \rangle$, and $s_1 \xrightarrow{b}_i s_2$ is shorthand for $s_1 \xrightarrow{b}_{\Pi[i]} s_2$;*
- *$\mathcal{V}$ is a finite set of synchronisation rules. A synchronisation rule is a tuple $\langle \overline{t}, a \rangle$, where a is an action label, and $\overline{t}$ is a vector of size n called a* synchronisation vector*, in which for all $i \in 1..n$, $\overline{t}[i] \in \mathcal{A}_i \cup \{\bullet\}$, where $\bullet$ is a special symbol denoting that $\Pi[i]$ performs no action.*

With $\mathcal{A}_{1..n}$, we refer to the union of the $\mathcal{A}_i$. Furthermore, for $\langle \overline{t}, a \rangle$, $Ac(\overline{t}) = \{i \mid i \in 1..n \wedge \overline{t}[i] \neq \bullet\}$ refers to the set of processes active for $\langle \overline{t}, a \rangle$, and $A(\overline{t}) = \{\overline{t}[i] \mid i \in 1..n\} \setminus \{\bullet\}$ refers to the set of actions participating in $\langle \overline{t}, a \rangle$.

A network of LTSs $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ is an implicit description of all possible system behaviour of the model. We call the explicit description the *system LTS* $\langle \mathcal{S}_\mathcal{M}, \mathcal{A}_\mathcal{M}, \mathcal{T}_\mathcal{M}, \mathcal{I}_\mathcal{M} \rangle$. It can be obtained by combining the $\Pi[i]$ according to the rules in $\mathcal{V}$:

- $\mathcal{I}_\mathcal{M} = \{\langle s_1, \ldots, s_n \rangle \mid \forall i \in 1..n. s_i \in \mathcal{I}_i\}$;
- $\mathcal{A}_\mathcal{M} = \{a \mid \langle \overline{t}, a \rangle \in \mathcal{V}\}$;
- $\mathcal{S}_\mathcal{M} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$;
- $\mathcal{T}_\mathcal{M}$ is the smallest transition relation satisfying: $\langle \overline{t}, a \rangle \in \mathcal{V} \wedge \forall i \in 1..n. (\overline{t}[i] = \bullet \wedge s'[i] = s[i]) \vee (\overline{t}[i] \neq \bullet \wedge s[i] \xrightarrow{\overline{t}[i]}_i s'[i]) \implies s \xrightarrow{a}_\mathcal{M} s'$.

On the left of Figure 1, a network consisting of three process LTSs and four synchronisation rules is shown. The black states are initial. The network LTS representing the behaviour of this network is shown on the right. The figure demonstrates the expressiveness of networks of LTSs. It shows, for example, that multi-party synchronisation is offered, as illustrated with the synchronisation rule $\langle \langle f, f, f \rangle, f \rangle$. This rule specifies that action f in the system LTS is the result of the synchronisation of the actions f of the three processes. Rule $\langle \langle b, d, \bullet \rangle, e \rangle$ specifies a synchronisation between processes $\Pi[1]$ and $\Pi[2]$, rule $\langle \langle a, \bullet, \bullet \rangle, a \rangle$ specifies that action a of process $\Pi[1]$ can be executed independently, and rule $\langle \langle \bullet, c, \bullet \rangle, c \rangle$ specifies the same for action c of process $\Pi[2]$.

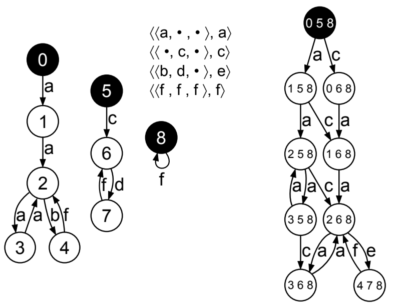


Fig. 1. A network of LTSs and its system LTS

Synchronisation rules can also be used to introduce non-deterministic behaviour, by specifying multiple rules for the same actions. By adding the rule $\langle\langle a, c, \bullet\rangle, g\rangle$, $\Pi[1]$ and $\Pi[2]$ can either synchronise on a and c, or perform them independently.

Hiding. To abstract from certain actions in networks, we define the hiding operator τ_H, which renames all actions in action set H, i.e. the *hiding set*, to τ: $\tau_H(\mathcal{M}) = \langle \Pi, \{(\langle \bar{t}, a\rangle \in \mathcal{V} \mid a \notin H\} \cup \{\langle \bar{t}, \tau\rangle \mid \langle \bar{t}, a\rangle \in \mathcal{V} \wedge a \in H\}\rangle$. Intuitively, hidden behaviour should neither be subjected to synchronisation, nor renamed: $\forall \langle \bar{t}, a\rangle \in \mathcal{V}. \tau \in A(\bar{t}) \implies |Ac(\bar{t})| = 1 \wedge a = \tau$. Hidden behaviour should also always be enabled: $\forall i \in 1..n. \exists \langle \bar{t}, \tau\rangle \in \mathcal{V}. A(\bar{t}) = \{\tau\} \wedge Ac(\bar{t}) = \{i\}$. We only consider networks for which these conditions hold.

Maximal Hiding. Mateescu and Wijs [21] explained how to derive for LTS $\mathcal{G}$ and temporal logic formula φ the *largest possible* hiding set $h_{\mathcal{A}_\mathcal{G}}(\varphi)$, if φ is written in a fragment of the modal μ-calculus [17] called L_μ^{dsbr}. Hiding this set, i.e. applying maximal hiding, allows moving to the highest possible level of abstraction without disturbing the truth-value of φ. L_μ^{dsbr} can express safety, liveness, and fairness properties. We denote the maximally hidden LTS $\mathcal{G}$ w.r.t. φ by $\tilde{\tau}_\varphi(\mathcal{G})$.

Divergence-Sensitive Branching Bisimulation. To compare LTSs, we use the equivalence relation *divergence-sensitive branching bisimulation* (DSBB) [11]. It supports hidden behaviour, and is sensitive to the branching structure of an LTS, including τ-cycles. Hence besides safety properties, it also supports fairness (e.g. livelock), and liveness properties. We call a state s *diverging*, iff an infinite τ-path is reachable from s. For finite LTSs, this means that a τ-cycle is reachable via τ-transitions. A formal definition of DSBB is not required for the understanding of this paper; the interested reader is referred to [11].

DSBB is compatible with maximal hiding. This allows reasoning about LTSs w.r.t. properties. Given a L_μ^{dsbr} formula φ and LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$, if we know that $\mathcal{G}_1$ satisfies φ, we can conclude whether or not $\mathcal{G}_2$ satisfies φ by applying maximal hiding on both LTSs, and determining whether $\tilde{\tau}_\varphi(\mathcal{G}_1)$ is DSBB to $\tilde{\tau}_\varphi(\mathcal{G}_2)$. Based on this, our proposed preservation check, formulated in Section 4, determines whether an LTS *transformation* application possibly alters the branching structure of an LTS. If not, we can conclude that φ is preserved after transformation.

3 LTS Transformations

In this section, we formalise refinement steps as transformations of networks of LTSs. A network is transformed by transforming the individual process LTSs that constitute it and adding additional synchronisation rules.

3.1 Transformation Rules

LTSs are transformed by applying transformation rules. These rules are defined as follows.

Definition 2. *A* transformation rule $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ *consists of a left pattern LTS* $\mathcal{L}^r = \langle \mathcal{S}_{\mathcal{L}^r}, \mathcal{A}_{\mathcal{L}^r}, \mathcal{T}_{\mathcal{L}^r}, \mathcal{I}_{\mathcal{L}^r} \rangle$ *and a right pattern LTS* $\mathcal{R}^r = \langle \mathcal{S}_{\mathcal{R}^r}, \mathcal{A}_{\mathcal{R}^r}, \mathcal{T}_{\mathcal{R}^r}, \mathcal{I}_{\mathcal{R}^r} \rangle$, *with* $\mathcal{I}_{\mathcal{L}^r} = \mathcal{I}_{\mathcal{R}^r} = (\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r})$.

States $\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}$, also referred to as the glue-states, are all initial and define how $\mathcal{R}^r$ should replace $\mathcal{L}^r$. All changes to an LTS are applied relative to these glue-states. We call a rule $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ applicable on an LTS $\mathcal{G}$ iff there exists a match $m_r : \mathcal{S}_{\mathcal{L}^r} \hookrightarrow \mathcal{S}_{\mathcal{G}}$ (an embedding) for which the following holds:

Definition 3. *A transformation rule* $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ *has a match* $m_r : \mathcal{S}_{\mathcal{L}^r} \hookrightarrow \mathcal{S}_{\mathcal{G}}$ *on an LTS* $\mathcal{G} = \langle \mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}} \rangle$ *iff* m_r *is injective and*

1. $\forall s_1 \xrightarrow{a}_{\mathcal{L}^r} s_2 . m_r(s_1) \xrightarrow{a}_{\mathcal{G}} m_r(s_2)$;
2. $\forall s \in \mathcal{S}_{\mathcal{L}^r} \setminus \mathcal{S}_{\mathcal{R}^r}, p \in \mathcal{S}_{\mathcal{G}}$:
   - $m_r(s) = p \implies \neg\exists s' \in \mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r} . m_r(s') = p$;
   - $m_r(s) \xrightarrow{a}_{\mathcal{G}} p \implies \exists s' \in \mathcal{S}_{\mathcal{L}^r} . s \xrightarrow{a} s' \wedge m_r(s') = p$;
   - $p \xrightarrow{a}_{\mathcal{G}} m_r(s) \implies \exists s' \in \mathcal{S}_{\mathcal{L}^r} . s' \xrightarrow{a} s \wedge m_r(s') = p$.

The second point of Definition 3 expresses the *gluing conditions* [14]. The first condition, the *identification condition*, says that in a single match, there may not be a contradiction concerning the removal of states, which could happen if both a glue-state and a non-glue-state are matched on the same state. The remaining two points express the *dangling condition*, which rules out the removal of transitions in $\mathcal{G}$ that are not explicitly represented in $\mathcal{L}^r$, i.e. it is not allowed that only one state of a transition is matched on and scheduled to be removed. We follow the double-pushout approach (DPO) [7], i.e. if the gluing condition is violated, the match is not valid. If we would apply transformation on a match even though the condition is violated, then the effect would be unpredictable, which would also limit our ability to reason about the structure of the result.

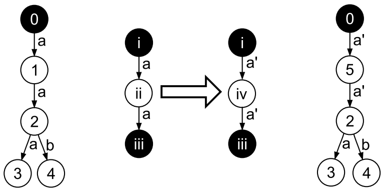


Fig. 2. Rule matching

In the middle of Figure 2, a transformation rule is shown. All initial and glue-states are coloured black in this figure. The rule defines that any state matched on state *ii* of the left pattern of the rule should be removed and replaced by a new state, which is labeled *iv* in the rule. Therefore, the left pattern can be matched on states $\{0, 1, 2\}$ of the LTS on the left of the figure, but not on states $\{1, 2, 3\}$. The latter match would result in the removal of state 2 and lead to a dangling transition.

Transformation of a network of LTSs proceeds as follows: First, the largest set of matches for a rule on each process LTS is determined. Then, for each match, DPO is applied to replace left pattern matches by copies of the right pattern, i.e. first remove all states and transitions matched by $\mathcal{L}^r \setminus \mathcal{R}^r$, and then place a copy of $\mathcal{R}^r \setminus \mathcal{L}^r$ in the result. Using this approach, termination of transformation is no issue; we do not recompute the set of matches for intermediate transformation results, and since the $\Pi[i]$ are finite, there is a finite number of matches initially.

Figure 2 illustrates the application of a transformation rule. The LTS on the right is the result of applying the rule in the middle to the LTS on the left.

3.2 Rule Systems

With transformation rules, a *rule system* $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ can be built, with R a set of transformation rules and $\hat{\mathcal{V}}$ a set of synchronisation rules to be introduced in the result of a transformation. Transformation of a network of LTSs via a rule system is done by determining for every $\Pi[i]$ and every rule the set of all matches, and applying transformation on these matches.

Here, a rule system defines how a network of LTSs should be transformed into a more refined network. In that context, it is important that a rule system is confluent, i.e. application always produces the same result. When a user defines a transformation, she desires to obtain a single, refined model. From graph theory, it is known that confluence is undecidable for general rule systems, but it is decidable under certain conditions [18]. Here, we ensure that a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ is confluent for an LTS $\mathcal{G}$ by 1) requiring that the action sets of left patterns of rules are disjoint, i.e. $\forall r_1, r_2 \in R. \mathcal{A}_{\mathcal{L}^{r_1}} \cap \mathcal{A}_{\mathcal{L}^{r_2}} = \emptyset$, and 2) checking for each $\Pi[i]$ that no two matches of a single rule intersect, i.e. $\forall r \in R. \neg \exists m_r^1, m_r^2, s_1, s_2 \in \mathcal{S}_{\mathcal{L}^r}. m_r^1 \neq m_r^2 \wedge m_r^1(s_1) = m_r^2(s_2)$. By 1) and the dangling condition, transformation of a match of one rule cannot influence a match of another rule, and by 2), neither can it influence the matches of the same rule. The first condition can efficiently be checked before matches are determined, and the second can be done while matches are determined. Note that these restrictions are more strict than technically required, but they still allow efficient confluence checking. If a rule system is not confluent, it often indicates that the user overlooked something; if not, then usually a confluent rule system can be obtained by e.g. rewriting actions, merging rule patterns, and / or splitting rule systems in multiple ones.

4 Checking Property Preservation

Our property preservation check for rule systems actually entails a number of computations and checks, which have been implemented in a new tool called Refiner. The only required input is a network of LTSs and a rule system, specified by the user. Figure 3 gives an overview of the approach.

Given $\mathcal{M}$ and Σ, the tool takes the following steps, which will be explained in more detail in this section:

1. Check that the new synchronisation rules $\hat{\mathcal{V}}$ are *well-formed* w.r.t. $\mathcal{M}$;
2. Generate a set Υ of *rule sets*;

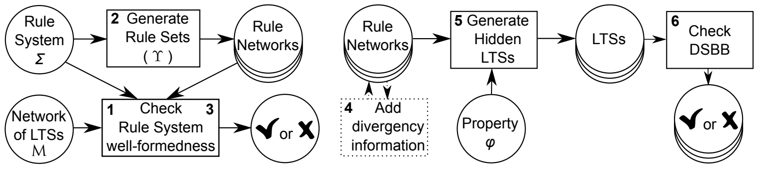


Fig. 3. Checking well-formedness and property preservation of a rule system

3. Check that Υ only contains *well-formed* sets w.r.t. $\mathcal{M}$;
4. Optionally, add divergency information to the rule sets;
5. For each rule set $\rho \in \Upsilon$, generate pairs of corresponding system LTSs and apply maximal hiding;
6. For each pair of LTSs, perform a DSBB comparison. If all DSBB comparisons in the previous step were positive, then Σ preserves φ.

We introduce two simplifications to facilitate explanation. First of all, we assume that in $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$, all the $\mathcal{A}_i$ are disjoint. This is not a fundamental limitation, since renaming of actions and modifying the synchronisation rules can enforce this. For a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$, this implies that each $r \in R$ can be applicable on at most one process LTS. If a similar transformation must be applied to multiple process LTSs, including in Σ multiple copies of a rule with appropriately renamed actions suffices. Based on this, we define a function $I : 2^{\mathcal{A}_{1..n}} \times \mathbb{N} \rightarrow 2^{\mathcal{A}_i}$, with, given an action set A, $I(A, i) = A \cap \mathcal{A}_i$.

Second of all, for $\mathcal{M}$ and Σ, we assume that each $\Pi[i]$ is matched on by exactly one r. This is expressed by indexing the $r \in R$ such that rule r_i is matched on $\Pi[i]$. This is also not a real limitation; if multiple rules are applicable on a $\Pi[i]$, Σ can be rewritten since it is confluent. This is done by splitting the rule system into multiple ones, and applying these one after the other.

1. Well-formedness of $\hat{\mathcal{V}}$*.* We restrict the ability to introduce new synchronisation rules in order to determine property preservation. Otherwise, by defining new synchronisation rules over already existing actions, a model could be altered without actually defining any transformation rules. We check that each rule in $\hat{\mathcal{V}}$ only contains actions in its vector that are introduced by Σ:

$$\forall \langle \bar{t}, a \rangle \in \hat{\mathcal{V}}, i \in 1..n. \bar{t}[i] \in (\mathcal{A}_{\mathcal{R}^{r_i}} \setminus \mathcal{A}_i) \cup \{\bullet\}$$

This does not limit the ability to express transformations, however; e.g. if two existing actions a and b should synchronise after transformation, one can define two transformation rules renaming these to a' and b', respectively, and define a new synchronisation rule for these new actions.

2. Generate Rule Sets. The synchronisation rules of $\mathcal{M}$ directly give rise to a dependency function δ for actions in the $\mathcal{A}_i$, where for each b, we have $\delta(b) = \bigcup_{\langle \bar{t}, a \rangle \in \mathcal{V}} \{\bar{t}[j] \mid j \in 1..n \wedge b \in \mathcal{A}_i \wedge \bar{t}[i] = b\} \setminus \{\bullet\}$. It defines the set of actions on which b depends to be able to synchronise in $\mathcal{M}$. This function can be used to identify the set of dependent actions containing the action set of the left pattern of an $r_i \in R$; it is the smallest closed set $C(\mathcal{A}_{\mathcal{L}^{r_i}})$ of $\mathcal{A}_{\mathcal{L}^{r_i}}$ w.r.t. δ and the subset relation, i.e. $\mathcal{A}_{\mathcal{L}^{r_i}} \subseteq C(\mathcal{A}_{\mathcal{L}^{r_i}})$ and for all $b \in C(\mathcal{A}_{\mathcal{L}^{r_i}})$, $\delta(b) \subseteq C(\mathcal{A}_{\mathcal{L}^{r_i}})$. This $C(\mathcal{A}_{\mathcal{L}^{r_i}})$ implies a set of dependent rules including r_i, namely $\rho_i = \{r_j \mid I(C(\mathcal{A}_{\mathcal{L}^{r_i}}), j) \neq \emptyset\}$. We define $\Upsilon = \{\rho_i \mid r_i \in R\}$.

Example 1. Consider Σ with $n = 4$ and for all $r_1, \ldots, r_4 \in R$, we have $\mathcal{L}^{r_i} = \langle \{s_i, s_i'\}, \{a_i\}, \{\langle s_i, a_i, s_i' \rangle\}, \{s_i\} \rangle$. We also have an $\mathcal{M}$ with $\mathcal{V} = \{\langle\langle a_1, a_2, a_3, \bullet\rangle, b\rangle, \langle\langle \bullet, \bullet, \bullet, a_4\rangle, a_4\rangle\}$. Now, $\rho_1 = \rho_2 = \rho_3 = \{r_1, r_2, r_3\}$ and $\rho_4 = \{r_4\}$.

3. Well-formedness of Rule Sets. One condition to check property preservation is that the $\rho \in \Upsilon$ are *complete* w.r.t. synchronising behaviour. In other words, we check that each action in $C(\mathcal{A}_{\mathcal{L}^{r_i}})$ is in the left pattern action set of some $r_i \in \rho$: $C(\mathcal{A}_{\mathcal{L}^{r_i}}) \subseteq \bigcup_{r_j \in \rho} \mathcal{A}_{\mathcal{L}^{r_j}}$. If this does not hold, then some relevant behaviour is not present in any of the left patterns, making it impossible to determine property preservation based on the rule patterns alone. Often, such a situation can be fixed by including rules for relevant behaviour that actually do not transform anything, i.e. the left pattern is equal to the right pattern.

Another condition concerns the applicability of Σ on $\mathcal{M}$: if rule r_i contains behaviour in its left pattern that requires synchronisation, then r_i must be applicable on all occurrences of that behaviour in $\Pi[i]$. We call this *universal applicability* of r_i on $\Pi[i]$ (note that action a requires synchronisation iff $|\delta(a)| > 1$, and that $\mathbf{ran}(m_{r_i})$ refers to the range of m_{r_i}):

$$\forall r_i \in R, a \in \mathcal{A}_{\mathcal{L}^{r_i}}.|\delta(a)| > 1 \implies \forall s_1 \xrightarrow{a}_i s_2.\exists m_{r_i}.\{s_1, s_2\} \subseteq \mathbf{ran}(m_{r_i})$$

Example 2. Consider the Σ of Example 1 again, but now without r_2. Then, $\rho_1 = \{r_1, r_3\}$ is not complete, since $C(\mathcal{A}_{\mathcal{L}^{r_1}}) = \{a_1, a_2, a_3\}$, and $a_2 \notin \mathcal{A}_{\mathcal{L}^{r_1}} \cup \mathcal{A}_{\mathcal{L}^{r_3}}$. The same holds for ρ_3.

Example 3. Consider an $\mathcal{M}$ with $n = 2$ and $\Pi[1]$ having the structure $p_1 \xrightarrow{a} p_2 \xrightarrow{a} p_3 \xrightarrow{b} p_4$. Furthermore, $\mathcal{V}$ contains a rule $\langle\langle a, c\rangle, d\rangle$. We also have a Σ containing r_1 with $\mathcal{L}^{r_1}$ having the structure $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3$. Now, r_1 is not universally applicable, since in $\Pi[1]$, not all occurrences of a are matched on by r_1. If instead, we had a synchronisation rule $\langle\langle a, \bullet\rangle, d\rangle$ in $\mathcal{V}$, then r_1 would be universally applicable, if b also requires no synchronisation.

4. Add Divergency Information. Transformation rules may introduce loops that after maximal hiding result in τ-loops. These may lead to a negative DSBB comparison result, since a non-diverging glue-state s in the left pattern of a rule can become diverging in the right. However, if the hidden system LTS of the input network already contains diverging behaviour, it could be the case that the matches of that rule only relate to those parts of the LTS where behaviour is already diverging, i.e. each process state p matched on by s is only part of diverging system state vectors in the system LTS. The introduction of additional

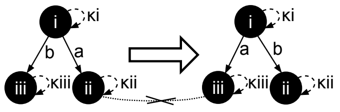


Fig. 4. An extended transformation rule

diverging behaviour will then not lead to a system LTS that is not DSBB to the original one. Such situations can be taken into account by first of all identifying which states are diverging in the hidden system LTS (this can be done in linear time with a slightly altered version of Tarjan's Strongly Connected Component detection algorithm [27]), propagating this information back to the process LTSs (a process state p is called diverging iff there exists no system state containing p that is non-diverging), and finally, adding a τ-selfloop to each s in the patterns of a rule if s is only matched on diverging process states.

Step 4 is optional, since it should only be done for transformation rules that do not *remove* diverging behaviour, since the added τ-loops will result in ignoring such removal. Removal of diverging behaviour can be detected by checking that each τ-loop in the left-pattern of a rule is represented in the right pattern.

5. Generate and Hide Relevant LTSs. To check property preservation of Σ, we need to make some structural information explicit in its rule patterns. If in a process LTS, a state is matched on by a glue-state, then it will remain in the LTS after transformation. This should be incorporated in a DSBB comparison. Consider the example in Figure 4. In this rule, the labels a and b are swapped between the transitions. Without the selfloops, a DSBB comparison will conclude that the two LTSs are equivalent, but it will not relate state *ii* (and *iii*) from the left pattern with state *ii* (and *iii*) from the right pattern, which indicates a structural change. To avoid such an erroneous conclusion, we introduce for each glue-state j a selfloop with a unique (fresh) label κ_j, and add a synchronisation rule to $\mathcal{V}$ stating that κ_j can be fired independently. Since only from state j action κ_j can be performed, both in the left and right pattern, a positive DSBB comparison outcome necessarily depends on being able to relate state j with itself. We refer with r_i^κ to rule r_i after application of the κ-modification.

Now, each $\rho \in \Upsilon$ directly defines two vectors $\overline{v}_\mathcal{L}$, $\overline{v}_\mathcal{R}$, where for $\mathcal{G} \in \{\mathcal{L}, \mathcal{R}\}$ and all $i \in 1..n$, we have $\overline{v}_\mathcal{G}[i] = \mathcal{G}^{r_i}$ if $r_i \in \rho$, and $\overline{v}_\mathcal{G}[i] = \langle \{s\}, \emptyset, \emptyset, \{s\} \rangle$ (a place-holder) otherwise. These vectors lead to two networks $\Xi_\mathcal{L}^\rho = \langle \overline{v}_\mathcal{L}, \mathcal{V} \rangle$ and $\Xi_\mathcal{R}^\rho = \langle \overline{v}_\mathcal{R}, \mathcal{V} \cup \hat{\mathcal{V}} \rangle$. The behaviour of $\Xi_\mathcal{R}^\rho$ represents the result in the system of applying the rule system to the behaviour of $\Xi_\mathcal{L}^\rho$.

Besides this pair of LTSs, we also derive LTSs for each non-empty subset of ρ, i.e. for all sets in $2^\rho \setminus \{\emptyset\}$. The subsets represent system states where some parties are able to perform synchronisation, whereas others may not be. The need to consider these states is illustrated by Example 4 described below.

6. DSBB Comparison of LTSs. For each $\rho \in \Upsilon$, we perform a DSBB comparison on all the $(\Xi_\mathcal{L}^{\rho'}, \Xi_\mathcal{R}^{\rho'})$, with $\rho' \in 2^\rho \setminus \emptyset$.

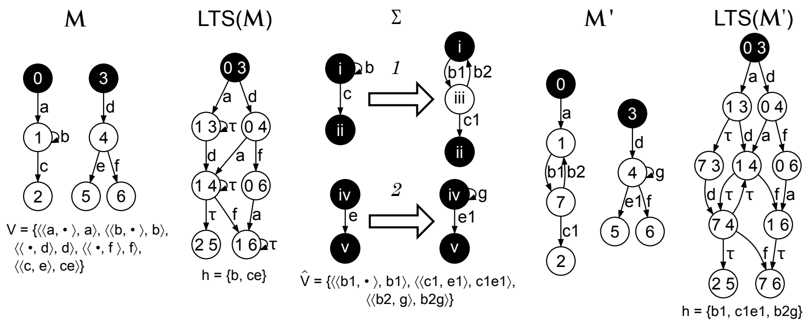

M
LTS(M)
Σ
M'
LTS(M')
V = {⟨⟨a, •⟩, a⟩, ⟨⟨b, •⟩, b⟩, ⟨⟨•, d⟩, d⟩, ⟨⟨•, f⟩, f⟩, ⟨⟨c, e⟩, ce⟩}
h = {b, ce}
V̂ = {⟨⟨b1, •⟩, b1⟩, ⟨⟨c1, e1⟩, c1e1⟩, ⟨⟨b2, g⟩, b2g⟩}
h = {b1, c1e1, b2g}

Fig. 5. The transformation of a network of LTSs

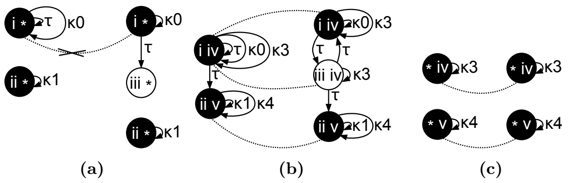


Fig. 6. DSBB comparisons of networks of (a) $\{1\}$, (b) $\{1,2\}$, and (c) $\{2\}$

Example 4. On the left of Figure 5, a network of LTSs $\mathcal{M}$ is shown together with its system LTS after hiding $h_{\mathcal{A}_{LTS(\mathcal{M})}}(\varphi) = \{b, ce\}$, which is the hiding set for some property φ. On the right, a rule system Σ consisting of two transformation rules and some synchronisation rules to be introduced at transformation is shown. Applying Σ on $\mathcal{M}$ results in network $\mathcal{M}'$ shown to the right of Σ with its system LTS after hiding $h_{\mathcal{A}_{LTS(\mathcal{M}')}}(\varphi) = \{b1, c1e1, b2g\}$. Transformation rules 1 and 2 are clearly dependent, since actions c and e in the left patterns must synchronise according to $\mathcal{V}$. Thus, we have $\rho_1 = \rho_2 = \{1,2\}$. In Figure 6b, the two networks of LTSs described by $\{1,2\}$ are compared after hiding the actions in h. The dotted lines in this figure illustrate that a DSBB exists for these two networks.

Even though a DSBB exists between these networks, the system LTSs of $\mathcal{M}$ and $\mathcal{M}'$ are not DSBB. This illustrates that it is not sufficient to only look at the combination of rule patterns. Instead, we also need to consider configurations in which some parties are able to perform synchronisation, whereas some parties are not. For example, the system LTS of $\mathcal{M}$ contains a state (1 3), which has a τ-loop that cannot be simulated by the system LTS of $\mathcal{M}'$. This τ-loop is the result of hiding the b-loop of state 1 in the leftmost process of $\mathcal{M}$. This process can perform action b independently. After transformation, however, a τ-cycle can only result from synchronisation between the transformed process LTSs of $\mathcal{M}'$ ($b2$ has to synchronise with g). In system states where this required synchronisation is impossible, as is the case in the system LTS of $\mathcal{M}'$ in state (1 3), only one τ-action can be performed. By considering 'subnetworks' of $\{1,2\}$, we detect this difference in the form of a negative DSBB comparison result (see Fig. 6a).

Correctness. Our check correctly determines whether a rulesystem is property preserving or not. This is proven in [8].

Complexity and Scalability. The bottleneck lies in computing the system LTS of a network, relating to state-space explosion. Most steps, however, do not involve this LTS. Only step 4 requires full analysis of the system LTS; however, divergency information can be propagated along property preserving transformations, i.e. it does not need to be recomputed for each new system LTS along a sequence of transformations. Given that we assume that the initial $\mathcal{M}$ can be verified, this does not introduce an additional time or space bottleneck.

Let k and m be the upper-bounds to the number of states and transitions, respectively, in a left or right rule pattern. Then, step 6 can be done for each ρ in $\mathcal{O}(2^{|\rho|} - 1 \cdot (k^{|\rho|} \cdot (k^{|\rho|} + m^{|\rho|}))$. Efficient DSBB detection takes $\mathcal{O}(k^{|\rho|} \cdot (k^{|\rho|} + m^{|\rho|}))$ [13], assuming the τ-loops have been compressed using Tarjan's algorithm, and there are $2^{|\rho|} - 1$ relevant subsets of ρ. An interesting observation is that the checks for the relevant subsets can be done fully independently, allowing for straightforward parallelisation. The space complexity of step 6 is $\mathcal{O}(k^{|\rho|} + m^{|\rho|})$.

In steps 5 and 6, worst-case, Σ would completely describe $\mathcal{M}$ in the left patterns of rules, and the right patterns would completely describe the refined model. In that case, the check actually boils down to generating the complete new LTS and checking if it is DSBB to the old one, which would not mitigate the state-space explosion problem. However, this would not be in line with the idea behind our technique. Typically, the rules in Σ contain patterns that are much smaller than the process LTSs. Then, the state spaces in step 5 will be exponentially smaller than the one of the transformed network.

Finally, our approach can only be successful if LTS transformation can be done efficiently; for a given rule, matching can be done linear to the size of the input LTS [6]. In our case, this is reasonable, as the process LTSs are usually exponentially smaller than the system LTS. Besides that, the use of transition labels means that we usually do not experience the worst-case complexity.

5 Experimental Results

Our check can be performed fully automatically by a new tool called REFINER, which integrates with the model checking toolsets CADP [10] and MCRL2 [12]; e.g., μ-calculus formulas can be verified using CADP, and in fact the MCRL2 tool *ltscompare* is used by REFINER to perform DSBB comparisons. REFINER has been implemented in Python, and can be run very efficiently using the Pypy interpreter[1]. Both REFINER and the CADP tool EXP.OPEN [19] can generate the system LTSs of networks; the latter is more efficient in doing so, but REFINER also stores how combinations of process states relate to the system states, which is required when computing divergency information in step 4 of our check.

We validated our approach using nine case studies on a machine with a quad-core INTEL XEON E5520 2.27 GHz processor, 1 TB RAM, running FEDORA 12.

[1] `http://www.pypy.org`

Table 1. LTS generation results

		LTS size (# states)	time (sec.)
ACS	$\mathcal{M}_0$	3,484	1.93
	$\mathcal{M}_1$	21,936	9.95
1394-fin	$\mathcal{M}_0$	198,692	13.95
	$\mathcal{M}_1$	6,679,222	305.02
wafer	$\mathcal{M}_0$	78,919	15.29
	$\mathcal{M}_1$	474,457	96.97
broadcast	$\mathcal{M}_0$	1,024	86.77
	$\mathcal{M}_1$	60,466,176	3486.76
	$\mathcal{M}_2$	60,466,176	4259.79
ABP	$\mathcal{M}_0$	759,375	29.97
	$\mathcal{M}_1$	380,204,032	26,509.29
	$\mathcal{M}_2$	656,356,768	56,365.93
HAVi-LE	$\mathcal{M}_0$	15,688,570	587.55
	$\mathcal{M}_1$	190,208,728	7,343.60
	$\mathcal{M}_2$	3,048,589,069	335,130.67
Sieve	$\mathcal{M}_0$	6,539,813	4,003.58
	$\mathcal{M}_1$	19,434,968	12,117.29
	$\mathcal{M}_2$	135,159,971	84,893.19
ODP	$\mathcal{M}_0$	91,394	26.73
	$\mathcal{M}_1$	7,699,456	117.13
DES	$\mathcal{M}_0$	64,498,297	771.26
	$\mathcal{M}_1$	64,498,317	814.20

Table 2. Preservation checking results

		hiding (sec.)	div. (sec.)	φ-pres. #	(sec.)	φ
ACS	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	0.26	0.37	56	10.26	✓
1394-fin	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	0.79	3.21	36	8.30	✓
wafer	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	0.85	1.57	17	3.21	✓
broadcast	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	0.48	1.07	4	0.90	✗
	$\mathcal{M}_0 \rightarrow \mathcal{M}_2$	-	-	70	21.10	✓
ABP	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	5.46	15.74	22	5.63	✗
	$\mathcal{M}_0 \rightarrow \mathcal{M}_2$	-	-	315	19.95	✓
HAVi-LE	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	325.52	690.28	127	39.74	✓
	$\mathcal{M}_1 \rightarrow \mathcal{M}_2$	-	-	31	6.02	✓
Sieve	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	85.77	215.00	51	45.86	✓
	$\mathcal{M}_1 \rightarrow \mathcal{M}_2$	-	-	51	25.25	✓
ODP	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	1.71	3.54	31	8.30	✓
DES	$\mathcal{M}_0 \rightarrow \mathcal{M}_1$	792.69	1468.86	3	255.14	✓

For each case study, we performed a number of refinements, and both verified the resulting system LTSs and checked property preservation of the refinements. We chose not to compare with other incremental approaches (see Section 6), because the latter support only transformations of 'flat' system LTSs, while we focus on refinements of individual process LTSs in a network. In particular, other approaches do not consider the interaction of processes in a system.

Table 1 displays the size in number of states and the verification time in seconds of the relevant system LTSs using Exp.Open, where $\mathcal{M}_0$ is the initial model. The first three cases stem from the set of examples distributed with the mCRL2 toolset, the last four are slightly altered versions of CADP models, and *ABP* and *broadcast* are two cases modelled by us. For each, we applied one of three different types of refinements to the process LTSs in their network: 1) adding non-synchronising transitions, representing additional internal computations or logging of messages (the first three and the last three cases), 2) adding support for lossy channels by introducing instances of the Alternating Bit Protocol (the ABP case), and 3) breaking down broadcast synchronisations into sequences of two-party synchronisations (the broadcast and the HAVi leader election case). All three types introduce new behaviour irrelevant for the property to be checked.

Table 2 displays the numbers related to preservation checking: per transformation, the time needed to hide irrelevant actions, the time needed to compute divergencies, and information related to the actual checking is shown. For the checking, the number of DSBB comparison checks, the total runtime, and the outcome of the check is displayed. The time needed for transformation is not displayed, but it takes at most as long as preservation checking, since the latter also involves rule matching. Note that hiding and divergency computation is only required before applying the first transformation on the initial model; for the same property, subsequent transformations can reuse the hidden network, and divergency information is updated when transforming. For the refinements of type 1, the standard preservation check sufficed, but for the other refinements, divergency information was required. The results clearly show the benefits of our approach: when the check concludes that a rule system preserves a property, exploration of the resulting system LTS can be avoided; this is fruitful if the transformation does not alter the LTS that much, as is the case for the DES protocol,[2] but the check really pays off when transformation leads to much larger LTSs. For example, in the HAVi leader election case, we have one subnetwork of three managers and one of three messaging systems, both of which involve three-party synchronisation. One practical refinement is to break these down into several two-party synchronisations, and in two transformation steps, this leads to models $\mathcal{M}_1$ and $\mathcal{M}_2$. Completely analysing the system LTS of $\mathcal{M}_2$ takes 93 hours, but the check can be done in about 6 seconds if hiding and divergency computation has already been done, and 17 minutes if this is not the case. In this case, we hid all behaviour irrelevant for a particular liveness property.

6 Related Work

Our work is related to incremental model checking. Early papers on this subject propose techniques to reuse model checking results of safety properties for a given LTS to determine whether it still satisfies the same property after some alterations [25,26]. Large speedups are reported compared to complete rechecking, but the memory requirements are at least as high, since all states plus additional bookkeeping per state must reside in memory. Our technique does not require this. Furthermore, we do not deal with large, flat LTSs directly, but with networks and transformation rules that both consist of relatively small LTSs. Finally, we do not recheck a property after transformation, but check bisimulation instead.

In the context of *Dynamic graph algorithms* [9], reachability is an *unbounded* problem [23,25], i.e. it cannot be determined solely based on the changes. Thanks to the gluing conditions and our criteria, this is not an issue in our context.

Saha [24] presents an incremental algorithm for updating bisimulation relations based on changes of a graph. The goal of Saha is to efficiently maintain a bisimulation, whereas the goal of our work is to assess whether a bisimulation is guaranteed to remain without actually constructing or maintaining it.

Work on finding refinement mappings, e.g. [1], is related, but the question whether there exists a mapping between two given models, establishing that one is an implementation of another, is different from having a model and a formalisation of how to transform it, and asking whether the transformation will preserve a property without looking at the application result. Work related to B, e.g. [20], is on strictly refining existing functionalities. We also support adding new functionality, as long as it is not relevant for the desired property.

Monotonically adding functionality, as opposed to refining, is addressed in e.g. [3]. The focus is on updating property formulae; it could be interesting to see if this is applicable in our setting to update properties.

[2] Note the long runtime of the divergency computation for the DES protocol, relative to generating its LTS with EXP.OPEN. Further improvement of the implementation of REFINER is expected to resolve this.

Combemale et al. [5], Hülsbusch et al. [15], and Karsai and Narayanan [16,22] check semantics preservation of model transformations using either strong or weak bisimilarity. They consider transformation to other modelling languages, whereas we focus on model refinement.

7 Conclusions and Future Work

We presented a technique aimed at verifying the correctness of complex models that are the result of iterative refinement through model transformation. It checks whether safety, liveness, and fairness properties are preserved by rule systems if they are well-formed w.r.t. the semantics of the input model. If a rule system preserves a property that holds for a given input model, construction and exploration of the new LTS can be avoided. Experiments show that preservation checking is several orders of magnitude faster than rechecking the property.

For future work, first, the concept of networks of LTSs could be extended to support additional features such as asynchronous communication. Furthermore, the relation between the formal notion of rule system and practical languages for the implementation of model transformations needs further study.

References

1. Abadi, M., Lamport, L.: The Existence of Refinement Mappings. Theoretical Computer Science 82, 253–284 (1991)
2. Beydeda, S., Book, M., Gruhn, V. (eds.): Model-Driven Software Development. Springer, Heidelberg (2005)
3. Braunstein, C., Encrenaz, E.: CTL-Property Transformations Along an Incremental Design Process. In: Proceedings of the Fourth International Workshop on Automated Verification of Critical Systems. Electronic Notes in Theoretical Computer Science, vol. 128, pp. 263–278. Elsevier (2004)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
5. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X.: Essay On Semantics Definition in MDE - An Instrumented Approach for Model Verification. Journal of Software 4(9), 943–958 (2009)
6. Dodds, M., Plump, D.: Graph Transformation in Constant Time. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 367–382. Springer, Heidelberg (2006)
7. Ehrig, H., Pfender, M., Schneider, H.: Graph Grammars: an Algebraic Approach. In: IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory, pp. 167–180. IEEE (1973)
8. Engelen, L.J.P., Wijs, A.J.: Checking Property Preservation of Refining Transformations for Model-Driven Development. CS-Report 12-08, Eindhoven University of Technology (2012)
9. Eppstein, D., Galil, Z., Italiano, G.: Dynamic Graph Algorithms. In: CRC Handbook of Algorithms and Theory of Computation, ch. 22. CRC Press (1997)
10. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)

11. van Glabbeek, R.J., Luttik, B., Trčka, N.: Branching Bisimilarity with Explicit Divergence. Fundamenta Informaticae 93(4), 371–392 (2009)
12. Groote, J.F., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y., van Weerdenburg, M., Wesselink, W., Willemse, T., van der Wulp, J.: The mCRL2 Toolset. In: Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques (2008)
13. Groote, J.F., Vaandrager, F.: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990)
14. Heckel, R.: Graph Transformation in a Nutshell. In: Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques. Electronic Notes in Theoretical Computer Science, vol. 148, pp. 187–198. Elsevier (2006)
15. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 183–198. Springer, Heidelberg (2010)
16. Karsai, G., Narayanan, A.: On the Correctness of Model Transformations in the Development of Embedded Systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop 2006. LNCS, vol. 4888, pp. 1–18. Springer, Heidelberg (2007)
17. Kozen, D.: Results on the Propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
18. Lambers, L., Ehrig, H., Orejas, F.: Efficient Detection of Conflicts in Graph-based Model Transformation. In: Proceedings of the International Workshop on Graph and Model Transformation. Electronic Notes in Theoretical Computer Science, vol. 152, pp. 97–109. Elsevier (2006)
19. Lang, F.: Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In: Romijn, J., Smith, G., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
20. Lano, K.: The B Language and Method, A Guide to Practical Formal Development. Springer, Heidelberg (1996)
21. Mateescu, R., Wijs, A.: Property-Dependent Reductions for the Modal Mu-Calculus. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 2–19. Springer, Heidelberg (2011)
22. Narayanan, A., Karsai, G.: Towards Verifying Model Transformations. In: Proceedings of the International Workshop on Graph Transformation and Visual Modeling Techniques. Electronic Notes in Theoretical Computer Science, vol. 211, pp. 191–200 (2008)
23. Ramalingam, G., Reps, T.: On The Computational Complexity of Dynamic Graph Problems. Theoretical Computer Science 158, 233–277 (1996)
24. Saha, D.: An Incremental Bisimulation Algorithm. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 204–215. Springer, Heidelberg (2007)
25. Sokolsky, O.V., Smolka, S.A.: Incremental Model Checking in the Modal Mu-Calculus. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 351–363. Springer, Heidelberg (1994)
26. Swamy, G.M.: Incremental Methods for Formal Verification and Logic Synthesis. PhD thesis, University of California (1996)
27. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)

Strength-Based Decomposition of the Property Büchi Automaton for Faster Model Checking

Etienne Renault[1,2], Alexandre Duret-Lutz[1], Fabrice Kordon[2], and Denis Poitrenaud[3]

[1] LRDE, EPITA, Kremlin-Bicêtre, France
[2] LIP6/MoVe, Université Pierre & Marie Curie, Paris, France
[3] LIP6/MoVe and Université Paris Descartes, Paris, France

Abstract. The automata-theoretic approach for model checking of linear-time temporal properties involves the emptiness check of a large Büchi automaton. Specialized emptiness-check algorithms have been proposed for the cases where the property is represented by a weak or terminal automaton.

When the property automaton does not fall into these categories, a general emptiness check is required. This paper focuses on this class of properties. We refine previous approaches by classifying strongly-connected components rather than automata, and suggest a decomposition of the property automaton into three smaller automata capturing the terminal, weak, and the remaining strong behaviors of the property. The three corresponding emptiness checks can be performed independently, using the most appropriate algorithm.

Such a decomposition approach can be used with any automata-based model checker. We illustrate the interest of this new approach using explicit and symbolic LTL model checkers.

1 Introduction

The automata-theoretic approach to linear-time model checking consists in checking the emptiness of the product between two Büchi automata: one automaton that represents the system, and the other that represents the negation of the property to check on this system.

There are many ways to apply this approach. *Explicit model checking* uses a graph-based representation of the automata. Usually the product is constructed on-the-fly as needed by the emptiness-check algorithm, and may be stopped as soon as a counterexample is found [7]. Additionally, partial-order reduction techniques can be used to reduce the state space [15]. *Symbolic model checking* uses a symbolic representation of automata, usually by means of decision diagrams [5]. In this approach the emptiness check is achieved using fixed points.

The run-time of these approaches can be improved by different means. One way is to optimize the property automaton by reducing its number of states or making it more deterministic, hoping for a smaller product with the system. Because the property automaton is small, the time spent optimizing it is negligible compared to the time spent performing the emptiness check of the product. Another possible improvement is to use an emptiness check algorithm tailored to the property automaton used. For instance generalized emptiness checks [19, 9] can be used when the property requires generalized acceptance conditions. Also, simplified procedures can be performed when the strength of the property automaton is weak or terminal [2, 6], improving the worst-case complexity by a constant factor.

For strong property automata (that are neither weak nor terminal), a general Büchi emptiness check algorithms has to be used, even though they could also contain some weak and terminal components. In this paper we focus such properties whose automata mix strong, weak, or terminal components. We show that such automaton can be decomposed into three automata, each of a different strength. These automata can then be emptiness checked independently (and concurrently) using the most appropriate algorithm. Each of these three automata is smaller than the original automaton, moreover, because it is simpler it can usually be even more simplified. This decomposition works regardless of the type model-checking approach and options used (explicit, symbolic, parallel,...).

This paper is organized as follows. In Section 2, we define the type of (generalized) Büchi automata we use, discuss their emptiness checks, and the hierarchy of automaton strengths. Section 3 studies different ways to characterize the strength of a strongly connected component. These strengths are the basis for our decomposition described in Section 4. Finally we present our experimental results in Section 5.

2 Büchi Automata and Their Strengths

Let AP be a finite set of (atomic) propositions, and let $\mathbb{B} = \{\bot, \top\}$ represent Boolean values. We denote $\mathbb{B}(AP)$ the set of all Boolean formulas over AP, i.e., formulas built inductively from the propositions AP, $\mathbb{B}$, and the connectives $\wedge$, $\vee$, and $\neg$. An assignment is a function $\rho : AP \to \mathbb{B}$ that assigns a truth value to each proposition. We denote $\mathbb{B}^{AP}$ the set of all assignments of AP.

The automata-theoretic approach is usually performed using Büchi automata. In this work, we use a slightly more general form of automata called *Transition-based Generalized Büchi Automaton* (TGBA) which allows a more compact representation of properties. Any Büchi automaton can be seen as a TGBA by pushing acceptance sets to outgoing transitions, so the reader working with Büchi automata will have no problem adapting our techniques.

Definition 1. *A **TGBA** is a 5-tuple $A = \langle AP, Q, q^0, \delta, F\rangle$ where:*
- *AP is a finite set of atomic propositions,*
- *Q is a finite set of states,*
- *$q^0 \in Q$ is the initial state,*
- *$\delta \subseteq Q \times \mathbb{B}^{AP} \times Q$ is the transition relation, labeling each transition by an assignment of the atomic propositions,*
- *$F \subseteq 2^{\delta}$ is a set of acceptance sets of transitions.*

A *run* of A is an infinite sequence of transitions $\pi = (s_1, \ell_1, d_1) \ldots (s_i, \ell_i, d_i) \ldots$ with $s_1 = q^0$ and $\forall i \geq 1$, $d_i = s_{i+1}$. Such a run is *accepting* iff it visits all acceptance sets infinitely often, i.e, $\forall f \in F$, $\forall i \geq 1$, $\exists j \geq i$, $(s_j, \ell_j, d_j) \in f$.

An infinite word $w = \rho_1\rho_2\cdots$ over $\mathbb{B}^{AP}$ (i.e., $\rho_i \in \mathbb{B}^{AP}$), is accepted by A iff there exists an accepting run $\pi = (s_1, \ell_1, d_1)\dots(s_i, \ell_i, d_i)\dots$ such that $\forall i, \rho_i = \ell_i$. The language $\mathscr{L}(A)$ is the set of infinite words accepted by A.

The automata-theoretic approach to model checking amounts to check the emptiness of the language of a TGBA that represents the product of a system (a TGBA where $F = \emptyset$) with the negation of the property to verify (another TGBA).

A *path* of length $n \geq 1$ between two states $q, q' \in Q$ is a finite sequence of transitions $\rho = (s_1, \ell_1, d_1)\dots(s_n, \ell_n, d_n)$ with $s_1 = q$, $d_i = q'$, and $\forall i \in \{1, \dots, n-1\}$, $d_i = s_{i+1}$. Let $S \subseteq Q$ such that $\{s_1, s_2, \dots, s_n, d_n\} \subseteq S$, we denote the existence of such a path by $q \overset{S}{\rightsquigarrow} q'$. If $q = q'$ we say that such a path is a *cycle*. A *cycle* is *accepting* iff it visits all acceptance sets, i.e., $\forall f \in F$, $\exists i \in \{1, \dots, n\}$, $(s_i, \ell_i, d_i) \in f$. A cycle is *elementary* iff it does not visit any state twice (i.e., $\forall 1 \leq i < j \leq n$, $s_i \neq s_j$).

If a TGBA has an (infinite) accepting run, then the run necessarily visits one of the states infinitely often, which means that the automaton has an accepting cycle that is reachable from q^0. One way to perform the emptiness check of a TGBA explicitly is therefore to search for such cycles using nested DFS (Depth First Search). Although there exists a nested DFS algorithm that works on TGBA [25], most of the usual nested DFS algorithms [23] require a degeneralized Büchi automaton with a single acceptance set (the degeneralization of a TGBA with n acceptance sets may multiply its number of states by n). In these algorithms, a first DFS is used to detect the start of potential cycles, and another (or several in the generalized case) DFS is started to detect an accepting cycle.

A second emptiness-check approach is to compute the accepting strongly-connected components of the TGBA.

Definition 2. *A **Strongly-Connected Component** (SCC) of a TGBA is a maximal set of states C such that there is a path between any two distinct states of C (i.e., $\forall s,\ s' \in C$, $(s \neq s') \Rightarrow (s \overset{C}{\rightsquigarrow} s')$).*

*C is **accepting** iff it contains an accepting cycle.*

*C is **complete** iff $\forall s \in C$, $\forall f \in \mathbb{B}^{AP}$, $\exists (q, \ell, q') \in \delta$ such that $s = q$, $f = \ell$, and $q' \in C$.*

While SCC-based emptiness checks [8, 16] are still based on a DFS exploration of the automaton, they do not require another nested DFS, and their complexity does not depend on the number of acceptance sets.

Symbolic emptiness checks [19, 14] are also based on the computation of SCCs in the symbolic representation of the automaton. This is done using fixed points on symbolic set of states, and amounts to performing a BFS-based emptiness check.

Whether based on nested DFS or SCC, explicit or symbolic, these emptiness-check procedures can be simplified according to the *strength* of the automaton representing the property to check [2, 13, 6, 23, 1].

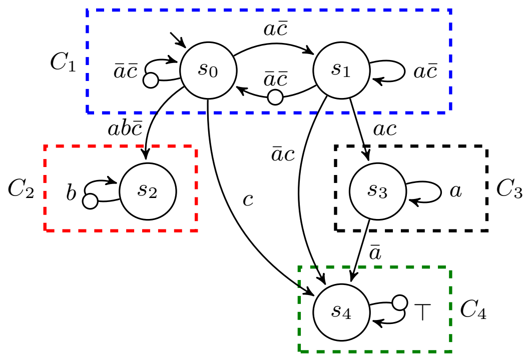


Fig. 1. TGBA for $(\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c$

Before defining the strength of the property automaton, let us first characterize the strength of an SCC.

Definition 3. *The strength of an SCC is:*
Non Accepting. *if it does not contain any accepting cycle,*
Inherently Terminal. *if it contains only accepting cycles and is complete,*
Inherently Weak. *if it contains only accepting cycles and it is not inherently terminal,*
Strong. *if it is accepting and contains some non-accepting cycle.*
These four strengths define a partition of the SCCs of an automaton.

There are two kinds of non accepting SCCs. If an SCC can only reach other non-accepting SCCs, it is **useless** and may be removed from the automaton without changing its language. This simplification is traditionally performed right after the translation of the property into an automaton. If the non accepting SCC can reach an accepting one, it is **transient**. In the rest this paper we assume that useless SCCs have been removed, i.e., all non-accepting SCCs are transient.

Figure 1 shows an example TGBA with a single acceptance set represented with white dots on transitions. Transitions are labeled by Boolean formulas instead of assignments (for instance a transition labeled by a is shorthand for two transitions labeled by ab and $a\bar{b}$). The dashed boxes highlight the five SCCs of the automaton. C_1 is a strong SCC (the cycle between s_0 and s_1 is accepting, while the self-loop on s_1 is a non-accepting cycle), C_2 is an inherently weak SCC, C_3 is transient, and C_4 is inherently terminal.

Definition 4. *An automaton is* ***inherently terminal*** *iff all its accepting SCCs are inherently terminal. An automaton is* ***inherently weak*** *iff all its accepting SCCs are inherently terminal or inherently weak. Any automaton is* ***general.*** *These three classes form a hierarchy where inherently terminal automata are inherently weak, which in turn are general.*

Note that the above constrains concern only accepting SCCs, but these automata may also contain non-accepting (transient) SCC.

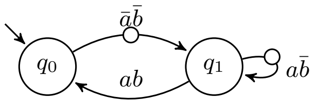


Fig. 2. An inherently weak automaton which is not weak

The notion of **inherently weak** automaton [3] generalizes the more common notion of **weak** automaton [2, 6]. If we define a weak SCC to be an accepting SCC whose transitions belong to all acceptance sets, then a weak automaton is an automaton that contains only weak, terminal, or non-accepting SCCs. A weak automaton is inherently weak, and an inherently weak automaton can be easily converted into a weak automaton [3]. For example the automaton from Fig. 2 can be easily converted into a weak automaton, by adding the transition from q_1 to q_0 into the ○ acceptance set.

Similarly, our definition of **inherently terminal** is a generalization of the notion of **terminal** automaton [2, 6]. If we define a terminal SCC to be weak and complete, then a terminal automaton should have only terminal, or non-accepting SCCs. A terminal automaton is inherently terminal, and an inherently terminal automaton can be obviously converted into a terminal automaton.

The emptiness-check algorithms previously discussed will obviously work with general automata. More efficient algorithms can be used for inferior strengths. For inherently weak automata, the explicit emptiness check reduces to the detection of a cycle in a inherently weak or terminal SCC. This can be performed using a single DFS [6]. Symbolic emptiness checks of inherently weak automata can be simplified similarly [2]. Furthermore, when the system to verify does not have any deadlock (each state has at least one successor) and the property automaton is terminal, then the emptiness check of the product becomes a reachability problem. Here again, both explicit and symbolic emptiness checks can take advantage of this simplification [2, 6].

Considering this strength hierarchy can also help when implementing techniques such as partial order reduction [6] or distributed model checking [1]. In most of the approaches suggested so far the improvements have only concerned (inherently) weak or terminal automata: if an automaton contains at least one strong SCC, a general emptiness check is required, even if it also contains SCCs of inferior strengths. However Edelkamp et al. [13] have suggested to consider the strengths of the SCCs to limit the scope of the nested DFS to the strong SCCs.

The technique we present in section 4 improves the emptiness check of properties that mix accepting SCCs of different strengths. A necessary step towards this goal is to be able to determine the strength of SCCs.

3 Determining SCC Strength

The SCCs of an automaton, and their acceptance, can be obtained by applying the algorithms of Couvreur [8] or Geldenhuys and Valmari [16].

We now consider three approaches to classify accepting SCCs. The **inherent approach**, that sticks to definition 3. A **structural heuristic**, based on the graph's structure. And a **syntactic heuristic**, which can only be applied when translation algorithm labels a state s of the automaton A by the LTL formula recognized from this state (this is the case in our implementation). The latter two heuristics may misclassify an SCC in a higher class, requiring a more general emptiness check algorithm.

We evaluate these three approaches on a benchmark of 10 000 random LTL formulas, translated into TGBA using Couvreur's algorithm [8] and where useless SCCs have been pruned. Couvreur's translation naturally outputs an inherently weak (resp. terminal) TGBA for any syntactic-persistence (resp. syntactic-guarantee) formula, in the syntactic classification of Černá and Pelánek [6]. For example, when translating the LTL formula $(\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c$, this translation produces the automaton from Fig. 1 in which states s_0, s_1, s_2, s_3, and s_4 respectively correspond to the LTL formulas $(\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c$, $\mathsf{F}\,\bar{a} \wedge ((\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c)$, $\mathsf{G}b$ (a syntactic-persistence formula), $\mathsf{F}\,\bar{a}$ and $\top$ (two syntactic-guarantee formulas).

We now describe how we characterize weak and terminal SCCs in the aforementioned three approaches.

If an accepting SCC contains any non-accepting cycle, then it necessarily contains a non-accepting elementary cycle. Therefore whether an accepting SCC is inherently weak can be determined by enumerating all its elementary cycles. As soon as one non-accepting cycle is found, the algorithm can claim the SCC to be non-inherently weak. This cycle enumeration can be costly since it may theoretically have to explore an exponential number of elementary cycles [20]. As an alternative, a structural heuristic, is to check whether all transitions in the accepting SCC belong to all acceptance sets (the SCC is weak), this information can be collected while we determine the accepting SCCs of the automaton. On our benchmark this approach correctly classifies $99,85\%$ of the weak SCCs. Another heuristic is to consider the LTL formulas labeling the states of the accepting SCC: if one of them is a syntactic-persistence then the SCC is either inherently weak or terminal. On our benchmark this test catches only $87,77\%$ of the weak SCCs.

Terminal SCCs can be similarly detected in three ways. The inherent approach is to check that (1) the disjunction of the labels of the outgoing transitions (that remain in the SCC) of each state is $\top$, and (2) there is no non-accepting elementary cycles. A structural heuristic would be to replace (2) by a check that all transitions belong to all acceptance sets. Finally, a syntactic heuristic would be to check that one state in the accepting SCC is labeled by a syntactic-guarantee formula. On our benchmarks these three approaches all catch 100% of the terminal SCCs.

The structural heuristics presented above correspond to the definition of the weak and terminal used by Bloem et al. [2] to characterize the strength of the entire automaton. Looking into the $0,15\%$ of SCCs that this structural heuristic fails to detect as inherently weak reveals that these SCCs are the results from the translation of pathological formulas: formulas whose syntactic class is above their actual strength. For instance $\varphi = \mathsf{G}(c \vee (\mathsf{X}\, c \wedge (\bar{c}\, \mathsf{U}\, b)))$ is a syntactic-recurrence formula equivalent to the safety formula $\mathsf{G}(c \vee (\bar{c} \wedge \mathsf{X}(c \wedge b)) \vee (b \wedge \mathsf{X}\, c))$, yet our translation of φ will produce an inherently weak automaton that is not weak.

In our experiments the structural approach was 3 times slower than the syntactic one, and 10 times faster than the inherent one. Since it caught $99,85\%$ of the weak SCCs, we adopted the structural approach in our upcoming experimentation. Regardless of these comparisons, all these approaches are instantaneous in practice.

Additional post-processing, as suggested by Somenzi and Bloem [24], would likely improve the "weakness" of the property automata.

4 Decomposing the Property Automaton According to Its SCCs Strengths

In this section, we focus on general property automata that cannot be handled by a specialized emptiness check (e.g. for inherently weak automata) because the property automaton contains SCCs of different strengths. The automaton from Fig. 1 is such an automaton. We show how they can be decomposed into three property automata representing their strong, weak, and terminal behaviors, that can be used concurrently.

We denote T, W, and S, the set of all transitions belonging respectively to some terminal, weak, or strong SCC. For a set of transitions X, we denote $\mathrm{Pre}(X)$ the set of states that can reach some transition in X. We assume that $q^0 \in \mathrm{Pre}(X)$ even if X is empty or unreachable.

Definition 5. *Let $A = \langle AP, Q, q^0, \delta, \{f_1, \ldots, f_n\}\rangle$ be a TGBA. We define three derived automata $A_T = \langle AP, Q_T, q^0, \delta_T, F_T\rangle$, $A_W = \langle AP, Q_W, q^0, \delta_T, F_W\rangle$, $A_S = \langle AP, Q_S, q^0, \delta_T, F_S\rangle$ that represent respectively the terminal, weak and strong behaviors of A, with:*

$$\begin{array}{lll} Q_T = \mathrm{Pre}(T) & F_T = \{T\} & \delta_T = \{(q, l, q') \in \delta \mid q, q' \in Q_T\} \\ Q_W = \mathrm{Pre}(W) & F_W = \{W\} & \delta_W = \{(q, l, q') \in \delta \mid q, q' \in Q_W\} \\ Q_S = \mathrm{Pre}(S) & F_S = \{f_1 \cap S, \ldots, f_n \cap S\} & \delta_S = \{(q, l, q') \in \delta \mid q, q' \in Q_S\} \end{array}$$

Fig. 3 shows the result of the decomposition of the TGBA of Fig. 1. The SCCs that are highlighted with boxes represent the terminal, weak, and strong SCCs that have been preserved. The rest of these automata is made of the prefixes leading to these accepting SCCs.

Property 1. The strengths of A_T, A_W, A_S are respectively terminal, weak, and strong (unless they have no transition).

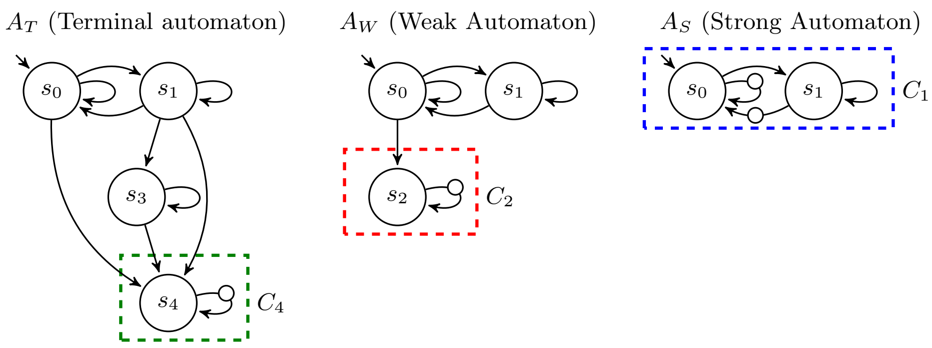


Fig. 3. Decomposition of the automaton from Fig. 1 into three automata (labels have been ommited for clarity)

Theorem 1. $\mathscr{L}(A) = \mathscr{L}(A_T) \cup \mathscr{L}(A_W) \cup \mathscr{L}(A_S)$.

Intuition of the proof: ($\subseteq$) A word accepted by A is recognized by a run that will eventually be captured by an accepting SCC of A. Since every accepting SCC belongs to one of the three automata, this SCC is necessarily reproduced in an accepting form in one of the three derived automata, and it is necessarily reachable from the initial state. ($\supseteq$) Because the three automata are restrictions of A, a word accepted by any of these is straightforwardly accepted by A.

Using this decomposition, we can perform three model-checking procedures in parallel, choosing the emptiness algorithm most suited to the strength of each derived automaton. This way, the more complex algorithm will have to deal with a smaller automaton (by construction), and the three procedures may abort as soon as one of them finds a counterexample.

The weak and terminal automata A_W and A_T, require very simple emptiness check algorithms [6, 2] because the acceptance conditions are easier to check. They also make it easier to apply other reduction such as partial order reductions [18], and they tend to produce smaller counterexamples [13].

For the strong derived automaton A_S, a general emptiness check is required. Implementations using an emptiness-check that can only deal with a single acceptance set (i.e., Büchi-style) need to degeneralize only this derived automaton.

This decomposition scheme can be further improved by minimizing each derived automaton. For instance, weak and terminal automata can be reduced very efficiently with techniques such as WDBA minimization [10]. Also simulations reductions [24] will be more efficient on automata with less acceptance conditions. As these techniques will not augment the strength of an automaton, they can be used without restriction.

In addition to reducing the number of states and acceptance sets in the automaton, the decomposition might also produce automata that observe fewer atomic propositions. Emptiness check techniques that are sensitive to the number of observed propositions [e.g., 22] will therefore benefit from the decomposition.

As a final note, this decomposition approach is suitable for any type of model checker (explicit, symbolic, parallel, ...) as long as it uses an automaton to represent the property.

5 Assessment

We compare the new decomposition approach against the classical one in four setups:

SE. This explicit setup uses Schwoon and Esparza's improved NDFS algorithm [23], to our knowledge, the best NDFS to date. This emptiness checks requires a degeneralization.

ELL. A refinement of the previous setup restricting the nested DFS to the strong components, as suggested by Edelkamp et al. [13].

Cou. This explicit setup uses Couvreur's SCC-based algorithm [8] and supports TGBA directly.

OWCTY. This symbolic setup uses an implementation of the classical OWCTY algorithm with multiple acceptance sets [19].

When the decomposition approach is used, the above emptiness checks are applied only on the strong automaton $K \otimes A_S$. For the products with weak and terminal automata, we use explicit or symbolic dedicated algorithms as described by Černá and Pelánek [6].

In all approaches, LTL formulas representing properties are first simplified, translated into TGBA, and these automata are postprocessed (using aforementioned techniques) in Spot [11]. In the decomposition scheme, the three resulting automata are postprocessed again.

The models we use come from the BEEM benchmark [21]. In explicit setup, we generate the system automaton K using a version of DiVinE 2.4 patched by the LTSmin team[1]. For the symbolic setup, we use a symbolic representation provided by `its-ltl`[2] [12].

Because the LTL formulas supplied by the BEEM benchmark are few and are usually safety automata (their negation translates into a terminal automaton), we opted to generate random LTL formulas for each model.

We ran our different approaches on 13 models, for which we selected formulas such (1) the property automaton contains different SCC strengths, (2) the product with the system has more than 2000 states, (3) for each model 100 formulas yield an empty product, and 100 formulas yield a non-empty one.[3] The second point is to avoid cases where the formula is trivial to verify.

[1] `http://fmt.cs.utwente.nl/tools/ltsmin/#divine`

[2] `http://ddd.lip6.fr/`

[3] This has been done by generating random formulas and running an emptiness check over the product automaton until 100 empty products and 100 non empty products were found. For a more detailed description of our setup, including selected models and formulas, see `http://move.lip6.fr/~Etienne.Renault/benchs/TACAS-2013/benchs.html`

Table 1. Sizes of the automata A_S, A_W, A_T relative to A, with or without the post-processing applied after decomposition, averaged on all our formulas

	no postproc.		postproc.	
	states	trans.	states	trans.
A_S	50.66%	37.87%	46.57%	34.85%
A_W	68.71%	51.47%	62.95%	44.77%
A_T	75.27%	63.68%	64.70%	49.28%

These tool chains were executed on a cluster of Intel Xeon E5645@2.40GHz, running Linux. The memory was confined to 4GB, and the run time to 1 hour.

Table 1 shows the reduction effect of the decomposition and additional post-processing on the sizes of the property automata. It can be noted that it is the strong automaton that obtains the greatest reduction, a good news, since this is the hardest to check.

Table 2 shows how many pairs of (model,formula) were successfully processed by each setup within the run-time and memory confinement. We separated empty products (verified formulas) from non-empty products (violated formulas) because the emptiness check may abort as soon as a counter example is found in the latter. It can be observed that using the decomposition always helps.

Table 2. Number of formulas processed by the classical (class.) and decomposition (dec.) approach, using different emptiness checks, out of a total of 2600 formulas

	empty		non-empty		total	
	class.	dec.	class.	dec.	class.	dec.
SE	1258	1297	1300	1300	2558	2597
ELL	1250	1297	1300	1300	2550	2597
Cou	1257	1299	1300	1300	2557	2599
OWCTY	1293	1299	1285	1299	2578	2598

Table 3 is an excerpt of our complete benchmark showing only a selection of the models whose verification required a significant run time (still, the observed trends are similar in other models). In order to compare the different algorithms, we restricted these measurements to formulas that could be processed by all setups.

For the "classical" explicit approaches, we measure the average number of visited states (counted once) and explored transitions (counted at most twice depending on the algorithm) during the emptiness check of $K \otimes A$ (the product of the system with A).

For the "decomposition-based" explicit approaches, three algorithms have been launched in parallel (on three different hosts) to check the emptiness of $K \otimes A_T$, $K \otimes A_W$, and $K \otimes A_S$. When $\mathscr{L}(K \otimes A) = \emptyset$, we have to wait for the three emptiness checks, and we report the performances of the last to terminate. When $\mathscr{L}(K \otimes A) \neq \emptyset$, we report the performance of the first emptiness check that finds a counterexample.

Table 3. Evaluation of the decomposition technique when model-checking different models in four possible setups. All values are averaged over all cases considered for one model. Time is in seconds, memory is in MB.

			classical				decomposition			
	model	algorithm	states	transitions	time	mem	states	transitions	time	mem
$\mathscr{L}(K \otimes A) = \emptyset$	at.4	SE	11778840	55765492	112.21	3034	7620732	30150665	63.35	2691
	84 cases	ELL	11778840	55748407	117.22	3050	7620732	30150665	63.07	2688
		Cou	11692421	54326243	95.95	2913	7542343	28859760	58.88	2657
		OWCTY			149.91	3227			75.68	2841
	bopdp.3	SE	2672100	14245549	20.59	1790	1430033	5249648	9.65	1460
	99 cases	ELL	2672100	13637796	21.27	1811	1440798	5250679	9.51	1443
		Cou	2515568	10389823	17.93	1717	1414104	4037319	7.96	1362
		OWCTY			241.26	3313			166.98	3151
	elevator2.3	SE	17583328	208607370	273.95	3622	12709300	106105555	161.48	3418
	64 cases	ELL	17583328	200251800	287.67	3639	12709300	106105555	161.02	3419
		Cou	17144611	171043227	186.22	3464	12479194	99666774	141.25	3348
		OWCTY			14.59	1607			6.48	1534
	elevator.4	SE	2928295	15794777	26.73	1969	1543723	4728263	10.58	1505
	94 cases	ELL	2928295	14908666	26.65	1984	1543723	4728263	10.54	1498
		Cou	2849219	12734156	20.14	1831	1547016	4430731	9.70	1463
		OWCTY			638.29	3812			245.55	3718
	prodcell.3	SE	3488725	25182172	34.28	1952	1358518	5065228	9.64	1400
	100 cases	ELL	3488725	23975933	35.11	1954	1358849	5065967	9.58	1397
		Cou	3194579	19584772	26.40	1797	1323029	4391328	8.38	1357
		OWCTY			145.60	3003			50.04	2731
$\mathscr{L}(K \otimes A) \neq \emptyset$	at.4	SE	362202	2384803	4.61	842	138	181	0.00	795
	93 cases	ELL	362202	2100874	4.38	861	146	186	0.00	798
		Cou	362196	2095924	3.63	837	172	217	0.00	799
		OWCTY			343.86	3501			80.95	2623
	bopdp.3	SE	32131	90859	0.19	765	1145	2333	0.01	803
	99 cases	ELL	31989	90668	0.20	762	1134	2310	0.01	802
		Cou	32120	80027	0.17	780	1152	2331	0.01	800
		OWCTY			292.19	3275			69.46	2594
	elevator2.3	SE	998871	14729965	15.29	1023	7967	50455	0.07	721
	100 cases	ELL	998725	13980443	16.54	1031	7978	50466	0.07	720
		Cou	984226	9916942	10.29	986	7975	50464	0.07	720
		OWCTY			30.53	2079			6.68	1172
	elevator.4	SE	37389	141012	0.28	745	54	58	0.00	719
	87 cases	ELL	37336	137843	0.29	751	44	47	0.00	718
		Cou	37386	118119	0.21	732	41	43	0.00	723
		OWCTY			491.27	3747			174.15	3087
	prodcell.3	SE	52458	313946	0.46	753	497	876	0.00	758
	97 cases	ELL	52375	271454	0.44	779	495	862	0.00	759
		Cou	48589	199349	0.32	744	491	857	0.00	757
		OWCTY			196.47	3209			57.83	2469

For all approaches (explicit and symbolic), we report peak memory usage and run time following the same rules as above.

A first observation is that while the run time is always improved by the decomposition, the memory gain is no always so obvious.

For non-empty products, the table shows that counterexamples are found much more rapidly. However when comparing the results of explicit approaches for non-empty products, we should keep in mind that there is a part of luck involved: depending on the order in which transitions of the property automaton are ordered, an emptiness check may find a counterexample faster. The results for empty products are easier to appreciate: since the entire product has to be explored transition order has no importance.

Table 3 can also be used as yet another comparison of emptiness check algorithms. We can notice that our benchmark favors explicit approaches over symbolic ones. This is a consequence of our selection of models and may certainly not be used to denigrate symbolic approaches. Still, if we order the emptiness check algorithm in the classical approach according to their average run time, we can observe that adding the decomposition does not change the order of these algorithms.

The ELL algorithm explores less transitions than SE because it restricts its nested DFS to the strong SCCs of the property, however this smaller exploration does not always reflect on the run-time because of the small overhead required to apply this restriction.

6 Conclusion

In the automata-theoretic approach to model checking for linear-time properties, specialized emptiness checks algorithms have been proposed for the cases where the property automaton is represented by a weak or terminal automaton. For strong automata, a general emptiness check is required.

In this paper we focused on properties whose automata (strong or weak) mix SCCs of different strengths, and for which we propose a decomposition approach based on these strengths.

Our experimentation of various ways to implement the characterization of SCC strengths has shown that trying to detect inherently weak SCCs (by enumerating all its elementary cycles) was not worth it: detecting weak SCCs is faster and easier to implement, and will miss very few inherently weak SCCs. However this study was performed on automata produced by Spot whose translation algorithm produce automata in the form preferred by the structural heuristic.

In the decomposition approach, instead of translating the property into one Büchi automaton A, we build three automata A_S, A_W, A_T of different strengths. These three automata are smaller than the original one, so checking them in parallel is necessarily faster. They also have a simpler structure, with less transitions in the acceptance sets of A_S and only one acceptance set for A_W and A_T, so they can be simplified more easily than A, improving the run time even more. Last but not least, more efficient algorithms are used for the emptiness check of $K \otimes A_W$ and $K \otimes A_T$.

Although we have experimented this approach with LTL formula, it will obviously work with any logic that can be translated into Büchi automata: for instance our implementation actually supports PSL. Similarly, we have experimented with some custom explicit and symbolic model checkers, but the same approach would be easily applied to any model checker based on the automata-theoretic approach. For instance we can decompose a property into three never claims to feed to the Spin model checker [17] and benefit from its partial-order reduction; or this approach could be integrated in VIS [4] and benefit from its SAT-based emptiness checks.

References

[1] Barnat, J., Brim, L., Ročkai, P.: On-the-fly parallel model checking algorithm that is optimal for verification of weak LTL properties. Science of Computer Programming 77(12), 1272–1288 (2012)

[2] Bloem, R., Ravi, K., Somenzi, F.: Efficient decision procedures for model checking of linear time logic properties. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 222–235. Springer, Heidelberg (1999)

[3] Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 611–625. Springer, Heidelberg (2001)

[4] Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G., Villa, T.: VIS: A System For Verification and Synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 428–432. Springer, Heidelberg (1996)

[5] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 1–33. IEEE Computer Society Press, Washington, D.C (1990)

[6] Černá, I., Pelánek, R.: Relating Hierarchy of Temporal Properties to Model Checking. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 318–327. Springer, Heidelberg (2003)

[7] Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithm for the verification of temporal properties. Formal Methods in System Design 1, 275–288 (1992)

[8] Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)

[9] Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-Fly Emptiness Checks for Generalized Büchi Automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)

[10] Dax, C., Eisinger, J., Klaedtke, F.: Mechanizing the Powerset Construction for Restricted Classes of ω-Automata. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 223–236. Springer, Heidelberg (2007)

[11] Duret-Lutz, A.: LTL translation improvements in Spot. In: Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2011). Electronic Workshops in Computing. British Computer Society, Tunis (2011), http://ewic.bcs.org/category/15853

[12] Duret-Lutz, A., Klai, K., Poitrenaud, D., Thierry-Mieg, Y.: Self-Loop Aggregation Product — A New Hybrid Approach to On-the-Fly LTL Model Checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 336–350. Springer, Heidelberg (2011)

[13] Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2-3), 247–267 (2004)

[14] Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is There a Best Symbolic Cycle-Detection Algorithm? In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 420–434. Springer, Heidelberg (2001)

[15] Geldenhuys, J., Hansen, H., Valmari, A.: Exploring the Scope for Partial Order Reduction. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 39–53. Springer, Heidelberg (2009)

[16] Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theoretical Computer Science 345(1), 60–82 (2005); Conference paper selected for journal publication

[17] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2003)

[18] Holzmann, G.J., Peled, D.A., Yannakakis, M.: On nested depth first search. In: Grégoire, J.C., Holzmann, G.J., Peled, D.A. (eds.) Proceedings of the 2nd Spin Workshop. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 32. American Mathematical Society (May 1996)

[19] Kesten, Y., Pnueli, A., Raviv, L.-O.: Algorithmic Verification of Linear Temporal Logic Specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 1–16. Springer, Heidelberg (1998)

[20] Loizou, G., Thanisch, P.: Enumerating the cycles of a digraph: A new preprocessing strategy. Information Sciences 27(3), 163–182 (1982)

[21] 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)

[22] Peled, D., Valmari, A., Kokkarinen, I.: Relaxed visibility enhances partial order reduction. Formal Methods in System Design 19(3), 275–289 (2001)

[23] Schwoon, S., Esparza, J.: A Note on On-the-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)

[24] Somenzi, F., Bloem, R.: Efficient Büchi automata for LTL Formulæ. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 247–263. Springer, Heidelberg (2000)

[25] Tauriainen, H.: Nested emptiness search for generalized Büchi automata. In: Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD 2004), pp. 165–174. IEEE Computer Society (June 2004)

Second Competition on Software Verification[⋆] (Summary of SV-COMP 2013)

Dirk Beyer

University of Passau, Germany

Abstract. This report describes the 2nd International Competition on Software Verification (SV-COMP 2013), which is the second edition of this thorough evaluation of fully automatic verifiers for software programs. The reported results represent the 2012 state-of-the-art in automatic software verification, in terms of effectiveness and efficiency, and as available and participated. The benchmark set of verification tasks consists of 2 315 programs, written in C, and exposing features of integers, heap-data structures, bit-vector operations, and concurrency; the properties include reachability and memory safety. The competition is again organized as a satellite event of TACAS.

1 Introduction

Software verification is a major research area within computer science, and modern implementations of verification tools become industrially relevant due to recent advancements in verification technology, e.g., new data structures for abstract domains and efficient solvers for satisfiability modulo theories (SMT). The competition on software verification systematically compares the effectiveness and efficiency of modern software verifiers. The community has gathered a benchmark set of a total of 2 315 verification tasks, which are arranged in eleven categories, according to the characteristics of the programs and the properties to verify. In difference to other competitions[1][2][3][4][5][6], SV-COMP [1] focuses on the evaluation of tools for *fully automatic* verification of *source code* programs in a standard programming language. All experiments are performed on dedicated competition machines with a resource specification that is the same for all participants. The goals of the competition on software verification are to:

- present the state-of-the-art in software-verification research,
- establish a widely accepted benchmark set of software verification tasks,
- make modern software verifiers visible, together with their strengths, and
- accelerate the transfer of new technologies to verification practice.

[⋆] http://sv-comp.sosy-lab.org
[1] http://www.satcompetition.org
[2] http://www.smtcomp.org
[3] http://ipc.icaps-conference.org
[4] http://www.qbflib.org/competition.html
[5] http://fmv.jku.at/hwmcc12
[6] http://www.cs.miami.edu/~tptp/CASC

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 594–609, 2013.

2 Procedure

The process of the competition consists of three phases: (1) *benchmark submission*, in which new verification tasks are collected and classified into competition categories (as in the first edition, all contributed benchmarks were accepted), (2) *training phase*, in which the benchmark set becomes frozen and teams of the competition candidates inspect the verification tasks and train their tools, (3) *evaluation phase*, in which all competition candidates were applied to the sets of verification tasks and the system descriptions were reviewed by the competition jury (all systems and their descriptions were archived and stamped with SHA hash values), and (4) *approval of verification results*, in which the teams received the preliminary results of their competition candidate. For more details on the procedure, we refer to the previous competition report [1].

3 Definitions and Rules

Verification Tasks. A verification task consists of a C program and a property. A verification run is a non-interactive execution of a competition candidate on a single verification task, in order to check if the following statement is correct: "The program satisfies the property." The result of a verification run is a triple (ANSWER, WITNESS, TIME). ANSWER is one of the following outcomes:

TRUE: The property is satisfied (no path that violates the property exists).
FALSE + Path: The property is violated (i.e., there exists a finite path that violates the property) and a counterexample path is produced and reported.
UNKNOWN: The tool cannot decide the problem, or terminates by a tool crash, or exhausts the computing resources time or memory (i.e., the competition candidate does not succeed in computing an answer TRUE or FALSE).

If the answer is FALSE, then a counterexample path must be produced and provided as WITNESS. There was so far no particular fixed format for the error path. The path has to be written to a file or on screen in a reasonable format to make it possible to check validity. TIME is the consumed CPU time until the verifier terminates. It includes the consumed CPU time of all processes that the verifier starts. If TIME is equal to or larger than the time limit, then the verifier is terminated and the ANSWER is set to 'timeout' (and interpreted as UNKNOWN). The C programs are partitioned into categories, which are defined in category-set files. The categories and the contained programs are explained on the benchmark page of the competition.

Property for Label Reachability. The property to be verified for all categories except 'MemorySafety' is the reachability property p_{error}, which is encoded in the program source code (using the error label 'ERROR'):

p_{error} **:** The C label 'ERROR' is not reachable from the entry of function 'main' on any finite execution of the program.

Table 1. Scoring schema for SV-COMP 2013

Reported result	Points	Description
UNKNOWN	0	Failure to compute verification result.
FALSE correct	+1	Violation of property in program was correctly found.
FALSE incorrect	−4	Violation reported for correct program (false alarm).
TRUE correct	+2	Correct program reported to satisfy property.
TRUE incorrect	−8	Incorrect program reported as correct (missed bug).

Property for Memory Safety. The property to be verified for category 'MemorySafety' is the memory-safety property $p_{\mathrm{mem-safety}}$, which consists of three partial properties:

$p_{\mathrm{mem-safety}}$ **:** $p_{\mathrm{valid-free}} \ \wedge\ p_{\mathrm{valid-deref}} \ \wedge\ p_{\mathrm{valid-memtrack}}$

$p_{\mathrm{valid-free}}$ **:** All memory deallocations are valid (counterexample: invalid free). More precisely: There exists no finite execution of the program from the entry of function 'main' on which an invalid memory deallocation occurs.

$p_{\mathrm{valid-deref}}$ **:** All pointer dereferences are valid (counterexample: invalid dereference). More precisely: There exists no finite execution of the program from the entry of function 'main' on which an invalid pointer dereference occurs.

$p_{\mathrm{valid-memtrack}}$ **:** All allocated memory is tracked, i.e., pointed to or deallocated (counterexample: memory leak). More precisely: There exists no finite execution of the program from the entry of function 'main' on which the program lost track of some previously allocated memory.

If a verification run detects that the property $p_{\mathrm{mem-safety}}$ is violated, the verification result is required to be more specific; the violated partial property has to be given in the result: FALSE(p), with $p \in \{p_{\mathrm{valid-free}}, p_{\mathrm{valid-deref}}, p_{\mathrm{valid-memtrack}}\}$, means that the (partial) property p is violated. The competition rules define that all programs in category 'MemorySafety' violate at most one (partial) property $p \in \{p_{\mathrm{valid-free}}, p_{\mathrm{valid-deref}}, p_{\mathrm{valid-memtrack}}\}$.

Benchmark Verification Tasks. All verification tasks are available for browsing and download via the public SVN repository of the Competition on Software Verification[7]. The programs were assumed to be written in GNU C (many of them adhere to ANSI C). Compared to SV-COMP 2012, it was not a requirement that the programs are provided in Cil (C Intermediate Language).

Evaluation by Scores and Run Time. Table 1 shows the scoring schema for SV-COMP 2013. Compared to the previous SV-COMP, the negative scores are doubled, in order to make it more difficult to compensate incorrect results by some correct results. The participating competition candidates are ranked according to the sum of points. Competition candidates with the same sum of points are further ranked according to success run time. The success run time for a competition candidate is the total CPU time over all verification tasks for which the competition candidate reported a correct verification result.

[7] `https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13`

As in the last edition, all verification runs were also performed on obfuscated versions of all benchmark programs (renaming of variable and function names; renaming of file name). There was no discrepancy between the verification results obtained from the obfuscated programs and the verification results obtained from the corresponding original program.

Opting-Out from Categories. Every team can submit for every category (including meta categories, i.e., categories that consist of verification tasks from other categories) an opt-out statement. In the results table, a dash is entered for that category; no execution results are reported in that category. If a team participates (i.e., does not opt-out), *all* verification tasks that belong to that category are executed with the verifier. The obtained results are reported in the results table; the scores for meta categories are weighted according to the established procedure. (This means, a tool can participate in a meta category and at the same time opt-out from a sub-category, with having the real results of the tool counted towards the meta category, but not reported for the sub-category.)

Computation of Score for Meta Categories. A meta category is a category that consists of several sub-categories. The score for such a meta category is computed from the normalized scores in its sub-categories. In SV-COMP 2013, there are two meta categories: ControlFlowInteger and Overall. ControlFlowInteger consists of the two sub-categories ControlFlowInteger-MemSimple and ControlFlowInteger-MemPrecise. Overall consists of the sub-categories BitVectors, Concurrency, ControlFlowInteger, DeviceDrivers64, FeatureChecks, HeapManipulation, Loops, MemorySafety, ProductLines, and SystemC.

The *score for a meta category* is computed from the scores of all k contained (sub-) categories using a normalization by the number of contained verification tasks: The normalized score sn_i of a verifier in category i is obtained by dividing the score s_i by the number of tasks n_i in category i ($sn_i = s_i/n_i$), then the sum $\Sigma_{i=1}^{k} sn_i$ over the normalized scores of the categories is multiplied by the average number of tasks per category.[8]

The goal is to reduce the influence of a verification task in a large category compared to a verification task in a small category, and thus, balance over the categories. Normalizing by score is not an option because we assigned higher positive scores for expected-true results and higher negative scores for incorrect results. (Normalizing by score would remove those desired differences.)

Competition Jury. The competition jury consists of the chair and one member of each participating team; the team-representing members circulate every year after the candidate-submission deadline. This committee reviews the competition contribution papers and helps the organizer with resolving any disputes that might occur. Table 2 lists the team-representing members of the jury of 2013.

[8] An example calculation can be found on the web page of the competition.

Table 2. Competition candidates with their system-description references and representing jury members

Competition candidate	Ref.	Jury member	Affiliation
Blast 2.7.1	[22]	Vadim Mutilin	Moscow, Russia
CPAchecker 1.1.10-Explicit	[19]	Stefan Löwe	Passau, Germany
CPAchecker 1.1.10-SeqCom	[24]	Philipp Wendler	Passau, Germany
CSeq 2012-10-22	[12]	Bernd Fischer	Southampton, UK
Esbmc 1.20	[20]	Lucas Cordeiro	Manaus, Brazil
Llbmc 2012-10-23	[11]	Carsten Sinz	Karlsruhe, Germany
Predator 2012-10-20	[10]	Tomas Vojnar	Brno, Czech Rep.
Symbiotic 2012-10-21	[23]	Jiri Slaby	Brno, Czech Rep.
Threader 0.92	[21]	Andrey Rybalchenko	Munich, Germany
Ufo 2012-10-22	[14]	Arie Gurfinkel	Pittsburgh, USA
Ultimate Automizer 2012-10-25	[15]	Matthias Heizmann	Freiburg, Germany

4 Participating Teams

This section briefly introduces the competition candidates (alphabetical order). Table 2 provides an overview of the participating candidates. Below, we list for each competition candidate the achieved (top-three) placements in the categories. The detailed summary of the results is presented in Sect. 5.

Table 3 provides an overview of the technologies and concepts that are used by the various competition candidates. The techniques of counterexample-guided abstraction refinement (CEGAR) [9], bounded model checking [8], and interpolation for discovering new facts to refine an abstract model [16] are used by a total of six tools. Other techniques that are offered by the competition candidates are predicate abstraction [13], construction of an abstract reachability graph (ARG) as proof of correctness [2], lazy abstraction [17], and shape analysis [18]. Only three tools support the verification of concurrent programs.

Blast 2.7.1 [2,22], submitted by *Pavel Shved, Mikhail Mandrykin, and Vadim Mutilin* (Russian Academy of Sciences, Russia), has achieved the placement:

- *Bronze* in DeviceDrivers64

Blast 2.7.1[9] is a software model checker that is based on predicate abstraction [13], CEGAR [9], and the interpolation tool CSIsat [7].

CPAchecker 1.1.10-Explicit [19], submitted by *Stefan Löwe* (University of Passau, Germany), has achieved the following placements:

- *Silver* in ControlFlowInteger
- *Silver* in DeviceDrivers64
- *Silver* in SystemC
- *Silver* in Overall

9 http://mtc.epfl.ch/software-tools/blast

Table 3. Technologies and features that the competition candidates offer

Competition candidate	**CEGAR**	**Predicate Abstraction**	**Bounded Model Checking**	**Shape Analysis**	**ARG-based Analysis**	**Lazy Abstraction**	**Interpolation**	**Concurrency Support**
BLAST	✓	✓			✓	✓	✓	
CPA-EXPLICIT	✓		✓		✓	✓	✓	
CPA-SEQCOM	✓	✓	✓		✓	✓	✓	
CSEQ			✓					✓
ESBMC			✓					✓
LLBMC			✓					
PREDATOR				✓				
SYMBIOTIC								
THREADER	✓	✓			✓		✓	✓
UFO	✓	✓	✓		✓	✓	✓	
ULTIMATE	✓	✓				✓	✓	

CPACHECKER 1.1.10-EXPLICIT is based on the verification framework CPACHECKER [4][10], which implements the formalism of configurable program analysis (CPA) [3]. The competition candidate uses an explicit-state model-checking approach [6] that integrates abstraction, CEGAR, and interpolation.

CPACHECKER 1.1.10-SEQCOM [24], submitted by *Philipp Wendler* (University of Passau, Germany), has achieved the following placements:

- *Winner* in Overall
- *Bronze* in BitVectors
- *Bronze* in ControlFlowInteger
- *Bronze* in FeatureChecks
- *Bronze* in HeapManipulation
- *Bronze* in ProductLines
- *Bronze* in SystemC

[10] `http://cpachecker.sosy-lab.org`

CPAchecker 1.1.10-SeqCom is also based on the verification framework CPAchecker and uses a sequential combination of an explicit-value analysis and predicate analysis with adjustable-block encoding [5].

CSeq 2012-10-22 [12], submitted by *Bernd Fischer, Omar Inverso, and Gennaro Parlato* (University of Southampton, UK), has achieved the placement:

– *Silver* in Concurrency

CSeq 2012-10-22[11] is an analyzer that transforms concurrent programs into non-deterministic sequential programs and starts a model-checking for sequential programs for the verification of the property on the resulting programs.

Esbmc 1.20 [20], submitted by *Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer* (University of Southampton, UK / UFAM, Brazil), achieved:

– *Silver* in BitVectors
– *Silver* in Loops
– *Bronze* in Concurrency
– *Bronze* in MemorySafety
– *Bronze* in Overall

Esbmc 1.20[12] is a bounded model checker that uses a combination of context-bounded symbolic model checking and k-induction for the verification of multi-threaded and single-threaded C programs.

Llbmc 2012-10-23 [11], submitted by *Stephan Falke, Florian Merz, and Carsten Sinz* (Karlsruhe Institute of Technology, Germany), has achieved the following placements:

– *Winner* in BitVectors
– *Winner* in Loops
– *Silver* in FeatureChecks
– *Silver* in Heapmanipulation
– *Silver* in MemorySafety
– *Silver* in ProductLines

Llbmc 2012-10-23[13] is a bounded model checker with a focus on a bit-precise analysis for low-level C code. The tool is based on the Llvm compiler infrastructure, and passes the verification conditions to the SMT solver Stp[14], which supports bit-vectors and arrays.

[11] `http://users.ecs.soton.ac.uk/gp4/cseq-0.1a.zip`
[12] `http://esbmc.org`
[13] `http://baldur.iti.uka.de/llbmc`
[14] `http://sites.google.com/site/stpfastprover`

Predator 2012-10-20 [10], submitted by *Kamil Dudka, Petr Muller, Petr Peringer, and Tomas Vojnar* (Brno University of Technology, Czech Republic), has achieved the following placements:

- *Winner* in FeatureChecks
- *Winner* in Heapmanipulation
- *Winner* in MemorySafety

Predator 2012-10-20[15] is a program analyzer focusing on the verification of C programs with dynamically-linked-list data structures. The abstract domain is based on symbolic memory graphs.

Symbiotic 2012-10-21 [23], submitted by *Jiri Slaby, Jan Strejcek, and Marek Trtík* (Masaryk University, Czech Republic), has participated in the categories ControlFlowInteger-MemPrecise, DeviceDrivers64, FeatureChecks, and SystemC. Symbiotic 2012-10-21 [16] is a program analyzer that combines instrumentation, program slicing, and symbolic execution.

Threader 0.92 [21], submitted by *Corneliu Popeea and Andrey Rybalchenko* (TU Munich, Germany), has achieved the following placements:

- *Winner* in Concurrency

Threader 0.92 [17] is a model checker for multi-threaded C programs that is based on compositional reasoning and supports, in addition to safety, also termination properties.

Ufo 2012-10-22 [14], submitted by *Arie Gurfinkel, Aws Albarghouthi, Sagar Chaki, Yi Li, and Marsha Chechik* (SEI, USA and University of Toronto, Canada), has achieved the following placements:

- *Winner* in ControlFlowInteger
- *Winner* in DeviceDrivers64
- *Winner* in ProductLines
- *Winner* in SystemC
- *Bronze* in Loops

Ufo 2012-10-22 [18] is a verifier that combines numerical data-flow domains with CEGAR, interpolation, and feasibility checks based on bounded model checking.

Ultimate Automizer 2012-10-25 [15], submitted by *Matthias Heizmann et al.* (University of Freiburg, Germany), has participated in the categories ControlFlowInteger-MemPrecise and SystemC. Ultimate Automizer[19] is a verifier that is based on trace abstraction, nested interpolants, and interpolation.

[15] http://www.fit.vutbr.cz/research/groups/verifit/tools/predator
[16] https://sf.net/projects/symbiotic
[17] http://www.model.in.tum.de/~popeea/research/threader.html
[18] https://bitbucket.org/arieg/ufo/wiki/Home
[19] http://ultimate.informatik.uni-freiburg.de/automizer

Table 4. Quantitative overview over all results — Part 1

Competition candidate Representing jury member Affiliation	**BitVectors** 60 points max. 32 verification tasks	**Concurrency** 49 points max. 32 verification tasks	**ControlFlowInteger** 146 points max. 94 verification tasks	**DeviceDrivers64** 2 419 points max. 1 237 verification tasks	**FeatureChecks** 206 points max. 118 verification tasks	**HeapManipulation** 48 points max. 28 verification tasks
Blast 2.7.1						
Vadim Mutilin	—	—	93	**2 338**	130	—
Moscow, Russia			7 100 s	2 400 s	42 s	
CPAchecker-Explicit						
Stefan Löwe	16	0	**143**	**2 340**	159	22
Passau, Germany	86 s	0 s	1 200 s	9 700 s	180 s	30 s
CPAchecker-SeqCom						
Philipp Wendler	**17**	0	**141**	2 186	**159**	**22**
Passau, Germany	190 s	0 s	3 400 s	30 000 s	160 s	29 s
CSeq 2012-10-22						
Bernd Fischer	—	**17**	—	—	—	—
Southampton, UK		270 s				
Esbmc 1.20						
Lucas Cordeiro	**24**	**15**	90	2 233	132	—
Manaus, Brazil	480 s	1 400 s	17 000 s	46 000 s	86 s	
Llbmc 2012-10-23						
Carsten Sinz	**60**	—	—	—	**166**	**32**
Karlsruhe, Germany	36 s				250 s	310 s
Predator 2012-10-20						
Tomas Vojnar	-75	0	-27	0	**166**	**40**
Brno, Czech Republic	95 s	0 s	650 s	0 s	6.0 s	2.3 s
Symbiotic 2012-10-21						
Juri Slaby	—	—	—	870	23	—
Brno, Czech Republic				230 s	11 s	
Threader 0.92						
Andrey Rybalchenko	—	**43**	—	—	—	—
Munich, Germany		570 s				
Ufo 2012-10-22						
Arie Gurfinkel	—	—	**146**	**2 408**	74	—
Pittsburgh, USA			450 s	2 500 s	46 s	
Ultimate 2012-10-25						
Matthias Heizmann	—	—	—	—	—	—
Freiburg, Germany						

Table 5. Quantitative overview over all results — Part 2

Competition candidate Representing jury member Affiliation	**Loops** 122 points max. 79 verification tasks	**MemorySafety** 54 points max. 36 verification tasks	**ProductLines** 929 points max. 597 verification tasks	**SystemC** 87 points max. 62 verification tasks	**Overall** 3 791 points max. 2 315 verification tasks
Blast 2.7.1					
Vadim Mutilin	35	—	652	34	80
Moscow, Russia	550 s		16 000 s	2 600 s	30 000 s
CPAchecker-Explicit					
Stefan Löwe	51	0	655	**61**	**2 030**
Passau, Germany	370 s	0 s	7 300 s	3 500 s	22 000 s
CPAchecker-SeqCom					
Philipp Wendler	50	0	**915**	**58**	**2 090**
Passau, Germany	1 400 s	0 s	3 100 s	1 800 s	41 000 s
CSeq 2012-10-22					
Bernd Fischer	—	—	—	—	—
Southampton, UK					
Esbmc 1.20					
Lucas Cordeiro	**94**	**3**	914	57	**1 919**
Manaus, Brazil	5 000 s	1 300 s	1 200 s	8 500 s	81 000 s
Llbmc 2012-10-23					
Carsten Sinz	**112**	**24**	**926**	49	—
Karlsruhe, Germany	540 s	38 s	3 600 s	1 900 s	
Predator 2012-10-20					
Tomas Vojnar	36	**52**	865	-6	799
Brno, Czech Republic	17 s	61 s	7 500 s	1 400 s	9 700 s
Symbiotic 2012-10-21					
Juri Slaby	—	—	—	0	—
Brno, Czech Republic				0 s	
Threader 0.92					
Andrey Rybalchenko	—	—	—	—	—
Munich, Germany					
Ufo 2012-10-22					
Arie Gurfinkel	**54**	—	**929**	**65**	-208
Pittsburgh, USA	750 s		5 000 s	3 000 s	12 000 s
Ultimate 2012-10-25					
Matthias Heizmann	—	—	—	45	—
Freiburg, Germany				4 800 s	

Table 6. Overview of the top-three verifiers for each category

Rank	Candidate	Score	Run Time	Solved Tasks	False Alarms	Missed Bugs
BitVectors						
1	**LLBMC 2012-10-23**	**60**	36	32		
2	ESBMC 1.20	24	480	27	3	2
3	CPACHECKER-SEQCOM	17	190	10		
Concurrency						
1	**THREADER 0.92**	**43**	570	28		
2	CSEQ 2012-10-22	17	270	11		
3	ESBMC 1.20	15	1 400	15	2	
ControlFlowInteger						
1	**UFO 2012-10-22**	**146**	450	94		
2	CPACHECKER-EXPLICIT	143	1 200	92		
3	CPACHECKER-SEQCOM	141	3 400	91		
DeviceDrivers64						
1	**UFO 2012-10-22**	**2 408**	2 500	1 228		
2	CPACHECKER-EXPLICIT	2 340	9 700	1 180		
3	BLAST 2.7.1	2 338	2 400	1 188		
FeatureChecks						
1	**PREDATOR 2012-10-20**	**166**	6.0	98		
2	LLBMC 2012-10-23	166	250	98		
3	CPACHECKER-SEQCOM	159	160	94		
HeapManipulation						
1	**PREDATOR 2012-10-20**	**40**	2.3	24		
2	LLBMC 2012-10-23	32	310	20		
3	CPACHECKER-SEQCOM	22	29	13		
Loops						
1	**LLBMC 2012-10-23**	**112**	540	74		
2	ESBMC 1.20	94	5 000	74	1	2
3	UFO 2012-10-22	54	750	64	10	
MemorySafety						
1	**PREDATOR 2012-10-20**	**52**	61	35		
2	LLBMC 2012-10-23	24	38	21		
3	ESBMC 1.20	3	1 300	10	2	
ProductLines						
1	**UFO 2012-10-22**	**929**	5 000	597		
2	LLBMC 2012-10-23	926	3 600	595		
3	CPACHECKER-SEQCOM	915	3 100	583		
SystemC						
1	**UFO 2012-10-22**	**65**	3 000	51		
2	CPACHECKER-EXPLICIT	61	3 500	44		
3	CPACHECKER-SEQCOM	58	1 800	42		
Overall						
1	**CPACHECKER-SEQCOM**	**2 090**	41 000	1 987		4
2	CPACHECKER-EXPLICIT	2 030	22 000	1 872		4
3	ESBMC 1.20	1 919	81 000	2 094	22	16

5 Results and Discussion

The results in this competition report represent the 2012 state-of-the-art in software verification in terms of effectiveness and efficiency, as available and participated. All presented results were approved by the competing teams.

The verification runs of the competition were (natively) executed on a dedicated unloaded compute server with a 3.4 GHz 64-bit Quad Core CPU (Intel i7-2600K) and a GNU/Linux operating system (x86_64-linux). The machine had 16 GB of RAM, of which exactly 15 GB were made available to the competition candidate. Every verification run had a run-time limit of 15 min. The run time in the tables is given in seconds of CPU time and all measurement values are rounded to two significant digits. One complete competition run of all candidates on all verification tasks required a total of 21 days of non-stop machine time; several such competition runs were necessary.

Tables 4 and 5 show the total quantitative overview. The tools are listed in alphabetical order. In every table cell for competition results, we list the score in the first row and the CPU time for successful runs in the second row. The top-three candidates are indicated by having their score formatted in bold face and in larger font size. The entry '—' means that the competition candidate opted-out from the category. For the calculation of the score and for the ranking, the scoring schema in Table 1 was applied.

Table 6 gives an overview of the top-three candidates for each category. The run time is given in seconds of CPU usage for the verification tasks that were successfully solved. The column 'False Alarms' indicates the number of verification tasks for which the tool reported an error but the program was correct (false positive), and column 'Missed Bugs' indicates the number of verification tasks for which the verifier claims that the program fulfills the property although it actually contains a bug (false negative).

Score-Based Quantile Functions for Quality Assessment. A total of six verifiers participated in the category Overall, for which we can discuss the overall performance over all categories together. (Note that the scores are normalized as described in Sect. 3.) Figure 1 illustrates the competition results using the quantile functions over all benchmark verification tasks. The function graph for a competition candidate yields, with each data point (x, y), the maximum run time y for the n fastest correct verification runs with the accumulated score x of all incorrect results and those n correct results.

This new visualization is helpful in analyzing the different aspects of verification quality, as outlined in the following.

Amount of Successful Verification Work. Results for verification tasks have different value, depending on the 'difficulty' of the verification task and on the correctness of the verification answer. This value is modeled by a community-agreed scoring schema (cf. Table 1). The x-width of a graph in Fig. 1 illustrates the value (amount) of successful verification work that the verifier has done. The verifier UFO 2012-10-22 is the best verification tool in this respect, because its

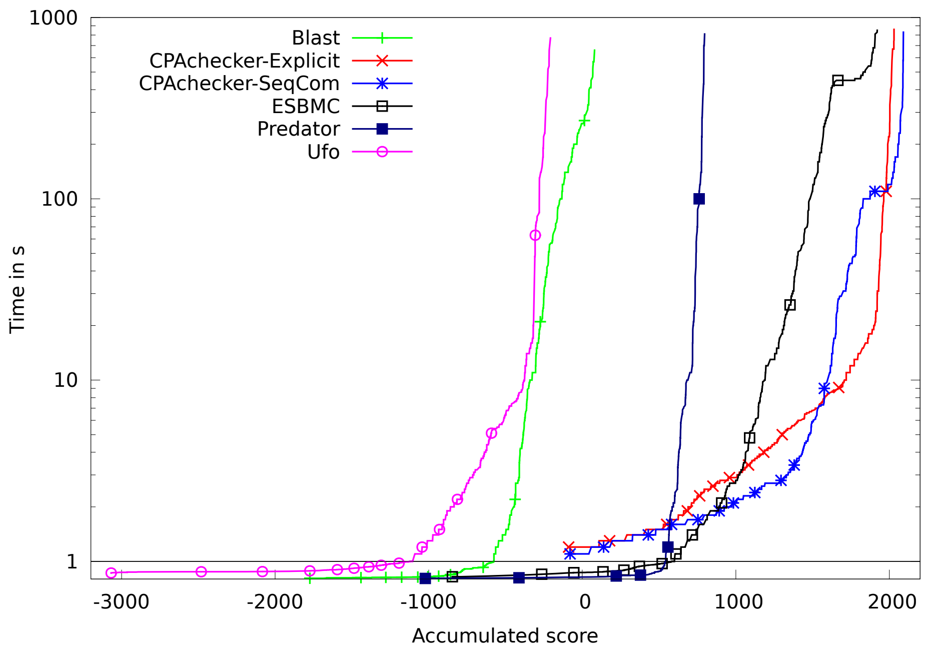


Fig. 1. Quantile functions: For each competition candidate, we plot all data points (x, y) such that the maximum run time of the n fastest correct verification runs is y and x is the accumulated score of all incorrect results and those n correct results. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s. The graphs are decorated with symbols at every 15-th data point.

quantile function has the largest x-width. This tool solved the most verification tasks, as also witnessed by the large score entries in Tables 4 and 5.

Amount of Incorrect Verification Work. The left-most data point yields the total negative score of a verifier (x-coordinate), i.e., the total score resulting from incorrect verification results. The more right a verifier starts its graph, the less incorrect results it has produced. The two CPAchecker-based candidates start with a very low negative score, and thus, have computed the least value of incorrect results, as also witnessed by the entry for category Overall in Table 6: the verifiers reported wrong results for only 4 out of 2 315 verification tasks.

Overall Quality Measured in Scores. The x-coordinate of the right-most data point of each graph represents the total score of the verification work (and thus, the total value) that was completed by the corresponding competition candidate. This measure identifies the winner of category Overall, as also reported in Table 6 (the x-coordinates match the score values in the table).

Characteristics of the Verification Tools. The y-coordinate of the left-most data point indicates the verification time for the "easiest" verification task for the verifier, and the y-coordinate of the right-most data point indicates the maximal time that the verifier spend on one single successful task (this is mostly just below the time limit). The area below a graph is proportional to the accumulated run time for all successfully solved verification tasks. Also the shape of the graph can give interesting insights: for example, the graphs for CPAchecker-SeqCom and Esbmc 1.20 show the characteristic bend that occurs if a verifier, after a certain period of time (100 s for CPAchecker-SeqCom and 450 s for Esbmc 1.20), performs a sequential restart with a different strategy.

Robustness, Soundness, and Completeness. The best tools of each category witness that today's verification technology has significantly progressed in terms of overall robustness (avoiding incorrect results), soundness (avoiding false negatives; missed bugs), and completeness (avoiding false positives; false alarms). The last two columns of Table 6 indicate the number of false alarms and missed bugs, respectively, for the top-three verifiers in each category.

6 Conclusion

The second edition of the competition on software verification was again well received in the research community. The participation increased from ten to eleven teams, the benchmark categories increased from seven to eleven, and the total number of benchmarks increased significantly to 2 315 verification tasks, of which 1 805 are expected to be correct, 492 contain a reachable error location, and 18 contain a violation of a memory-safety property. The organizer and the jury were making sure that the competition follows the high quality standards of the TACAS conference, in particular to respect the important principles of fairness, community support, transparency, and technical accuracy.

The results witness a significant progress of the state-of-the-art in developing new concepts for verification of software and in advancing the tool implementations that fully automatically perform the verification. The participating verification tools were able to verify the majority of verification tasks. The top verifiers are quite reliable in the categories that they are focusing on, in terms of robustness, soundness, and completeness. There is no single technique that is superior to all others. The competition candidates —SMT-based model checkers, bounded model checkers, explicit-state model checkers, and program analyzers— showed their different, complementing strength in the various categories.

Acknowledgement. We thank Karlheinz Friedberger for his support during the evaluation phase and for his work on the benchmarking infrastructure.

References

1. Beyer, D.: Competition on Software Verification (SV-COMP). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker Blast. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)

3. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program Analysis with Dynamic Precision Adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)
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. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
6. Beyer, D., Löwe, S.: Explicit-State Software Model Checking Based on CEGAR and Interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013)
7. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
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. 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)
10. Dudka, K., Müller, P., Peringer, P., Vojnar, T.: Predator: A Tool for Verification of Low-level List Manipulation (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 629–631. Springer, Heidelberg (2013)
11. Falke, S., Merz, F., Sinz, C.: LLBMC: Improved Bounded Model Checking of C Programs using LLVM (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 625–628. Springer, Heidelberg (2013)
12. Fischer, B., Inverso, O., Parlato, G.: CSeq: A Sequentialization Tool for C (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 618–620. Springer, Heidelberg (2013)
13. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
14. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: UFO: Verification with Interpolants and Abstract Interpretation (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 639–642. Springer, Heidelberg (2013)
15. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 643–645. Springer, Heidelberg (2013)
16. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: Proc. POPL, pp. 232–244. ACM (2004)
17. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
18. Jones, N.D., Muchnick, S.S.: A Flexible Approach to Interprocedural Data-Flow Analysis and Programs with Recursive Data Structures. In: POPL, pp. 66–74 (1982)
19. Löwe, S.: CPAchecker with Explicit-Value Analysis Based on CEGAR and Interpolation (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 612–614. Springer, Heidelberg (2013)

20. Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Handling Unbounded Loops with ESBMC 1.20 (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 621–624. Springer, Heidelberg (2013)
21. Popeea, C., Rybalchenko, A.: Threader: A Verifier for Multi-threaded Programs (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 635–638. Springer, Heidelberg (2013)
22. Shved, P., Mandrykin, M., Mutilin, V.: Predicate Analysis with BLAST 2.7 (Competition Contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
23. Slaby, J., Strejček, J., Trtík, M.: Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 632–634. Springer, Heidelberg (2013)
24. Wendler, P.: CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis (Competition Contribution). In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 615–617. Springer, Heidelberg (2013)

CPAchecker with Explicit-Value Analysis Based on CEGAR and Interpolation (Competition Contribution)

Stefan Löwe

University of Passau, Germany

Abstract. CPAchecker is a freely available software-verification framework, built on the concepts of Configurable Program Analysis (CPA). Within CPAchecker, several such CPAs are available, e.g., a Predicate-CPA, building on the predicate domain, as well as an Explicit-CPA, in which an abstract state is represented as an *explicit* variable assignment. In the CPAchecker configuration we are submitting, the highly efficient Explicit-CPA, backed by interpolation-based counterexample-guided abstraction refinement, joins forces with an auxiliary Predicate-CPA in a setup utilizing dynamic precision adjustment. This combination constitutes a highly promising verification tool, and thus, we submit a configuration making use of this analysis approach.

1 Software Architecture

CPAchecker is designed as an extensible framework for software verification, which is written in Java. The framework allows for parsing the input program into its internal data structures and provides interfaces to SMT solvers and interpolation procedures (e.g., MathSAT[1]). The paramount design decision of CPAchecker is separation of concerns, thus, each of the ready-made verification algorithms available within CPAchecker is implemented as a single CPA [1]. As these CPAs may be flexibly recombined on a per-demand basis, developing novel verifiers or reusing existing components for other domains is greatly facilitated.

2 Verification Approach

CPAchecker [2] represents the set of reachable states as an abstract reachability graph (ARG), which is built by successor computations along the edges of the program's control-flow automaton. The nodes of the ARG, representing sets of reachable program states, track all the relevant information, such as the program counter, the call stack, and the abstract data states of the main CPAs.

In contrast to our contribution from last year, which was doing software model checking via predicate abstraction, the main CPA in our configuration for this year, namely the Explicit-CPA, performs explicit-state software model checking

[1] http://mathsat4.disi.unitn.it/

in order to verify properties of a program. This approach has the advantage over more sophisticated techniques, like, e.g., approaches based on predicate abstraction, that the state representation is simpler and successor computation is more efficient. However, once applied to real-world code, representing each and every state of a program explicitly is bound to fall prey to the problem of state-space explosion, unless a proper abstraction technique is put in place. To this end, we extend the Explicit-CPA by abstraction and interpolation-based counterexample-guided abstraction refinement (CEGAR) [3].

There, the analysis starts with an initially empty precision, and, eventually, a counterexample will be found. The (in)feasibility of the (spurious) counterexample will be determined by a full-precision check, performed by our Explicit-CPA. If infeasible, the interpolation procedure will extract a refined, parsimonious precision to be used in the next iteration of the CEGAR loop. This abstract–refine approach circumvents the problem of state-space explosion in many cases, leading to improved run times and more solved instances than the naive approach.

However, due to the less expressive state representation of the explicit domain, it can occur that during explicit refinement, a spurious counterexample cannot be excluded by means of the explicit domain. To further improve the precision of our analysis, we add a Predicate-CPA in a dynamic precision adjustment approach [3]. There, the precision of the Predicate-CPA gets only refined in just those corner cases where the Explicit-CPA lacks expressiveness, and accordingly, the Predicate-CPA plays only an auxiliary, supporting role in the whole analysis.

Additionally, to limit the number of false positives reported by our verifier, once the analysis finds what it believes to be a real counterexample, the respective error path is given to CBMC.[2] Only if CBMC agrees with our result, the bug will be reported, otherwise, the analysis continues hunting for another bug.

3 Strengths and Weaknesses

The CPAchecker framework is striving for maximal reuse of existing components like the parser front-end, interfaces to the theorem provers, and, as described above, already existing CPAs. Hence, we rather adhere to software-engineering best practices instead of optimizing algorithms into a highly-tuned, but then also monolithic piece of software that becomes ever harder to maintain.

We expect our verifier to perform well where the property to be proven is strongly connected to the control flow. This is confirmed by the impressive results we obtained in the categories "ControlFlowInteger", "SystemC" and, most notably, in the category "DeviceDrivers64", where compared to the naive approach, our novel abstract–refine concept also has the most noticeable effect [3].

However, CPAchecker, and in particular the CPAs used in our configuration, do not provide support for the verification of properties in multi-threaded or recursive programs, while also lacking support for properties regarding memory safety. We wish to add more thorough handling of structures, unions, pointers, pointer aliasing and heap data structures into the analysis in the near future.

[2] http://www.cprover.org/cbmc

4 Setup and Configuration

CPAchecker is available under the Apache 2.0 license and both source code and binary releases are available for download at http://cpachecker.sosy-lab.org. Due to the fact that CPAchecker is written in Java, it is deployable on almost any platform. However, configurations depending on the predicate analysis currently work only under GNU/Linux, because the MathSAT library is available only for this platform. For the purpose of the software-verification competition, we submit version `1.1.10-svcomp13` of CPAchecker, with the configuration `sv-comp13--explitp-pred`. The command line for running this configuration is

```
./scripts/cpa.sh -sv-comp13--explitp-pred -heap 12000m
                 -disable-java-assertions path/to/sourcefile.cil.c
```

For C programs that assume a 64-bit environment (i.e., those in the category “Linux Device Drivers 64-bit”) the parameter stated below needs to be added:

```
-setprop cpa.predicate.machineModel=LINUX64
```

For the category “Memory Safety”, the property to verify is given by `-spec p` with `p` in {`valid-free`, `valid-deref`, `valid-memtrack`}. On machines with less than 16 GB RAM, we recommend to decrease the amount of memory given to the Java VM accordingly. CPAchecker will print the verification result and the name of the output directory to the console. Additional information, e.g., the error path, will be written to the respective files in this output directory.

5 Project and Contributors

The CPAchecker project is as an international open-source project, maintained at the University of Passau by the Software Systems Lab. It is used and extended by members of the Russian Academy of Science, the Technical University of Vienna, and the University of Paderborn. We would like to thank all contributors for their help and efforts spent on the CPAchecker project, and in particular, we would like to thank Dirk Beyer for maintaining the CPAchecker project.

References

1. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
2. 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)
3. Beyer, D., Löwe, S.: Explicit-Value Analysis Based on CEGAR and Interpolation. Technical Report MIP-1205, University of Passau/ArXiv 1212.6542 (2012)

CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis (Competition Contribution)

Philipp Wendler

University of Passau, Germany

Abstract. CPAchecker is an open-source framework for software verification, based on the concepts of Configurable Program Analysis (CPA). We submit a CPAchecker configuration that uses a sequential combination of two approaches. It starts with an explicit-state analysis, and, if no answer can be found within some time, switches to a predicate analysis with adjustable-block encoding and CEGAR.

1 Verification Approach

CPAchecker [3] is an open software-verification framework that integrates several state-of-the-art approaches for software model checking. None of these approaches is a clear winner over the other in terms of successfully verified programs and performance. Instead, each technique has its own distinct advantages and might solve programs that other analyses perhaps cannot verify. Thus we use a sequential combination of two analyses in order to be able to verify more programs than the two analyses alone. The second analysis is started after the first analysis if the first terminated with no verification result, e.g., because of an exhaustion of the available resources. This is a simple instance of conditional model checking [1] without information passing between the subsequent analysis runs. An overview of the approach can be seen in Fig. 1.

We use as a first analysis a rather simple explicit analysis which tracks values of integer variables. It does not use concepts such as CEGAR or lazy abstraction. This analysis often finds counterexamples quickly, but fails in other cases due to state-space explosion. In particular, we have experienced that it is unlikely to produce a result if it is not successful in short time. Thus we limit this analysis to a runtime of 100 s. If the analysis has neither found a valid counterexample nor proved the program safe after this time, it will terminate gracefully. In this case, we use the predicate analysis from last year's competition submission [5] as the second analysis. It uses lazy predicate abstraction with adjustable-block encoding [4], CEGAR, and Craig interpolation.

In order to prevent false alarms, we dump each counterexample in form of a (loop-free) C program, and run the bit-precise bounded model checker CBMC[1] in version 4.2 on it. If CBMC refutes the reachability of the error in the generated program, we skip the counterexample and continue with the analysis.

[1] http://www.cprover.org/cbmc

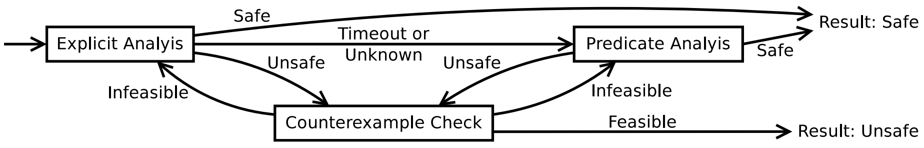


Fig. 1. Verification approach

2 Software Architecture

CPAchecker is written in Java and based on the Configurable Program Analysis (CPA) framework [2]. The explicit analysis, the predicate analysis, and "helper" analyses that are used in this configuration, such as tracking of the program counter, call stack, and function pointers, are implemented as CPAs. CPAs can be enabled as desired without changing other CPAs, and are used by a common algorithm for reachability analysis. Other algorithms, for example for CEGAR, counterexample checks, and analysis combinations, wrap the core algorithm depending on the configuration. The framework also uses the C parser from the Eclipse CDT project[2], and MathSAT4[3] as an SMT solver and interpolation engine.

3 Strengths and Weaknesses

The main advantage of the submitted configuration is the combination of two conceptually different analyses. This allows verifying a wide variety of programs. For example, most programs in the category "ProductLines" can be verified by the rather simple explicit analysis in short time, whereas the predicate analysis fails on many of them. However, in cases where the explicit analysis is not successful, this combination may lead to a decreased performance. For example, the predicate analysis alone needed 1000 s for the category "ControlFlowInteger" in the 2012 competition, whereas now 3400 s are needed (obtaining the same score).

The precise counterexample checks with CBMC make sure that CPAchecker never produces a wrong counterexample. Furthermore, CPAchecker only reports 4 programs erroneously as safe. No other tool in the competition that participated in all categories managed to achieve a non-negative score in all categories.

The implementation of CPAchecker sticks closely to the theoretical concepts and is primarily focused on re-usability and flexibility (witnessed by the existence of extensions contributed by other groups). The use of the CPA framework makes CPAchecker an easily extensible framework for software verification as it allows to re-use and combine analyses implemented as CPAs. However, this means that CPAchecker does not exploit all possible low-level optimizations that are available in a strongly coupled implementation.

[2] http://www.eclipse.org/cdt/

[3] http://mathsat4.disi.unitn.it/

Multi-threaded programs and the verification of memory-safety properties are currently not explicitly supported by CPAchecker.

4 Setup and Configuration

CPAchecker is available online at http://cpachecker.sosy-lab.org under the Apache 2.0 license. It requires Java 6 to run. We submitted CPAchecker in version `1.1.10-svcomp13` using the configuration `-sv-comp13--combinations` to the competition on software verification. The command line for running it is

```
./scripts/cpa.sh -sv-comp13--combinations -heap 12000m
                   -disable-java-assertions path/to/sourcefile.cil.c
```

For C programs that assume a 64-bit environment (i.e., those in the category "Linux Device Drivers 64-bit") the below parameter needs to be added:

```
-setprop cpa.predicate.machineModel=Linux64
```

For the category "Memory Safety", the property to verify is given by `-spec p` with `p` in {`valid-free`, `valid-deref`, `valid-memtrack`}. For machines with less RAM, the amount of memory given to the Java VM needs to be adjusted with the parameter `-heap`. CPAchecker will print the verification result and the name of the output directory to the console. Additional information (such as the error path) will be written to files in this directory.

5 Project and Contributors

CPAchecker is an open-source project lead by Dirk Beyer from the Software Systems Lab at the University of Passau. Several other research groups use and contribute to CPAchecker, e.g., the Russian Academy of Science, the University of Paderborn, and the Technical University of Vienna.

We would like to thank all contributors for their work on CPAchecker since 2007. The full list can be found online at http://cpachecker.sosy-lab.org.

References

1. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012)
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. 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. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
5. Löwe, S., Wendler, P.: CPAchecker with Adjustable Predicate Analysis (Competition Contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 528–530. Springer, Heidelberg (2012)

CSeq: A Sequentialization Tool for C
(Competition Contribution)

Bernd Fischer[1,2], Omar Inverso[1], and Gennaro Parlato[1]

[1] Electronics and Computer Science, University of Southampton, UK
[2] Department of Computer Science, Stellenbosch University, South Africa
{b.fischer,oi2c11,gennaro}@ecs.soton.ac.uk

Abstract. Sequentialization translates concurrent programs into equivalent non-deterministic sequential programs so that the different concurrent schedules no longer need to be handled explicitly. It can thus be used as a *concurrency pre-processor* for many sequential program verification techniques. CSeq implements sequentialization for C and uses ESBMC as sequential verification backend [5].

1 Introduction

Sequentialization is a recent verification technique that translates a concurrent program into a non-deterministic sequential program that (under certain assumptions) behaves equivalently, so that the different concurrent schedules do not need to be explicitly handled during verification. It can be implemented as a source-to-source program transformation and can be used as a *concurrency pre-processor* for sequential program verification tools, which in principle makes it an attractive and general approach.

However, in practice, only a few tools exist, and most of them work on an idealized language such as Boolean programs, or on an intermediate representation level, which makes them unsuitable as concurrency pre-processors for third-party tools. With CSeq, we aim to close this gap, and to develop a sequentialization tool for the full C language.

2 Verification Approach

Lal/Reps Sequentialization Schema. CSeq largely follows the schema proposed by Lal and Reps [7], which replaces the control non-determinism inherent to concurrent programs by data non-determinism. More specifically, it translates a concurrent program ($t_1 \ || \ t_2$) into a sequential but non-deterministic program ($t'_1 \ ; \ t'_2 \ ; \ c$), which contains additional copies of the shared global memory; c checks any assumptions on these copies that have been made independently by the transformed threads t'_1 and t'_2.

CSeq replaces each shared variable x by a k-indexed entry $x[k]$ in an array of size K where k is an auxiliary variable called the current round counter and K is the round bound. The transformed program then calls the thread functions sequentially, in the same order in which they are created. It simulates a context switch simply by non-deterministically increasing k up to the round bound K; if k grows beyond K, an early return is enforced (i.e., the thread is pre-empted). CSeq inserts this simulation code at all sequence points of the original program.

In this schema, the first thread accesses a fresh copy of the memory for each round, with non-deterministically chosen values, while the subsequent threads always continue with the state left by their predecessor at each round. The initial guesses are stored in a second copy $x'[k]$; at the end of the program c then checks that each round has ended with the guesses that are used in the next round, i.e., that $x[j] = x'[j+1]$ holds; simulations that do not satisfy this condition do not correspond to feasible runs, and are discarded.

Since infeasible runs are only discarded at the end, assertion and reachability checking need to be integrated with the sequentialization; in particular, in order to prevent false results, errors can only be reported after the checker c has run. CSeq thus replaces all assertions by conditionals that set an error variable that is tested by c. The same argument also applies to implicit safety properties such as array bounds violations, or nil pointer dereferences that are handled by the applied backend verification tool. In principle, these need to be translated into explicit checks, and their detection by the backend needs to be explicitly suppressed. However, CSeq does currently not support this.

Related Approaches. Sequentialization was originally developed for two threads and two context switches only by Qadeer and Wu [9], but was subsequently generalized by Lal and Reps to a fixed number of threads and a parameterized number of round-robin scheduling [7]. Later, LaTorre/Madhusadan/Parlato extended [7] to track only reachable configurations [10]. Further extensions allowed modelling of unbounded, dynamic thread creation [6,3,11], and dynamically linked data structures allocated on the heap [1]. Like CSeq, Rek [4] implements sequentialization for C via code-to-code transformation, but it is targeted at real-time systems and hard-codes a specific scheduling policy. Poirot [8] also verifies concurrent C programs via sequentialization, but it first translates them into Boogie and then implements the sequentialization transformation at the Boogie level.

3 Architecture, Implementation, and Availability

Architecture. CSeq is implemented as a source-to-source transformation tool in Python (v2.7.1). It uses the `pycparser` (v2.08) [2] to parse a C program into an abstract syntax tree (AST), and then traverses the AST to construct the sequentialized version, as outlined above. The result can then be processed by any verification tool for C; a small script (`cseq-esbmc`) bundles up translation and verification by ESBMC.

Availability and Installation. CSeq can be downloaded from `http://users.ecs.soton.ac.uk/gp4/cseq-0.1a.zip`. It can be installed as global Python script; it also requires installation of the `pycparser`, and ESBMC (v1.20, which is available at `www.esbmc.org`) must be on the path to use the `cseq-esbmc` script.

Call. For the competition, CSeq should be called in the installation directory as follows: `./cseq-esbmc <file>`.

Limitations. CSeq is in the initial development and there are still some limitations on the structure of the programs that can be translated, and on the properties that can be checked. Currently we assume that the `main` function consists of an initialization stage, in which the variables are initialized and a known number of threads is created, followed by a shut-down stage that includes all (if any) pthread_join's. We currently do not support conditional waiting nor pthread_join and pthread_exit with return variables. We implemented a deadlock check, but do not use it for the competition, as it is not required by the benchmarks. Since heap-allocated memory is accessible to all threads, it needs to be treated similarly to global variables; CSeq does not support this yet. Lifting these restrictions, and in particular supporting dynamic memory, dynamic thread creation, and conditional waiting will require significant efforts.

We further assume that the declarations for the global variables precede those for all functions, that there are no static variables and no global multi-dimensional arrays, and that local variables cannot shadow global variables. We do not support switch statements, due to limitations in the pycparser. These limitations simplified our implementation and can be lifted relatively easily.

Sequentialization is in principle independent of the verification tool used as back-end, but the current version of CSeq is (tightly) integrated with ESBMC. Despite this, ESBMC's counterexamples are not yet translated back into the original concurrent program, although this is a purely mechanic process.

4 Results

Since CSeq is a concurrency pre-processor, we only competed in the Concurrency category. Here, CSeq did well, and correctly solved 11 of the 33 benchmarks, with no false results, winning the Silver medal. In particular, it scored better than ESBMC v1.20 with its built-in concurrency handling. The existing implementation limitations showed particularly prominently in the CIL-preprocessed benchmarks, which CSeq could not handle.

References

1. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. Logical Methods in Computer Science 7(4) (2011)
2. Bendersky, E.: Pycparser, http://code.google.com/p/pycparser/
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. Chaki, S., Gurfinkel, A., Strichman, O.: Time-bounded analysis of real-time systems. In: FMCAD, pp. 72–80 (2011)
5. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE, pp. 331–340 (2011)
6. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: POPL, pp. 411–422 (2011)
7. 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)
8. Qadeer, S.: Poirot—A Concurrency Sleuth. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, p. 15. Springer, Heidelberg (2011)
9. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI, pp. 14–24 (2004)
10. 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)
11. La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing parameterized programs. In: FIT. EPTCS, vol. 87, pp. 34–47 (2012)

Handling Unbounded Loops with ESBMC 1.20 (Competition Contribution)

Jeremy Morse[1], Lucas Cordeiro[2], Denis Nicole[1], and Bernd Fischer[1,3]

[1] Electronics and Computer Science, University of Southampton, UK
[2] Electronic and Information Research Center, Federal University of Amazonas, Brazil
[3] Department of Computer Science, Stellenbosch University, South Africa
esbmc@ecs.soton.ac.uk

Abstract. We extended ESBMC to exploit the combination of context-bounded symbolic model checking and k-induction to prove safety properties in single- and multi-threaded ANSI-C programs with unbounded loops. We now first try to verify by induction that the safety property holds in the system. If that fails, we search for a bounded reachable state that constitutes a counterexample.

1 Overview

ESBMC is a context-bounded symbolic model checker that allows the verification of single- and multi-threaded C code with shared variables and locks. Previous versions of ESBMC can only be used to find property violations up to a given bound k but not to prove properties, unless we know an upper bound on the depth of the state space; however, this is generally not the case. In this paper, we sketch an extension of ESBMC to prove safety properties in bounded model checking (BMC) via mathematical induction. The details of ESBMC are described in our previous work [2–4]; here we focus only on the differences to the version used in last year's competition (1.17), and in particular, on the combination of the k-induction method with the normal BMC procedure.

2 Differences to ESBMC 1.17

Except for the loop handling described below, ESBMC 1.20 is largely a bugfixing version. The main changes concern the memory handling, the internal data structures (where we replaced CBMC's string-based accessor functions), and the Z3 encoding (where we replaced the name equivalence used in the pointer representation by the more appropriate structural equivalence). We have also changed our `pthread`-handling and added missing sequence points (most importantly at the `pthread_join`-function), which can lead to an increase in the number of interleavings to be explored. These changes lead to substantial improvements in robustness and speed, as a comparison of both versions over this year's benchmarks shows. Over the entire competition set of 2315 benchmarks and with an unwind bound of $n = 6$, ESBMC 1.20 produces 70 internal assertion violations, compared to a total of 405 for ESBMC 1.17. The new version also produces a smaller number of false results, leading to a score improvement of more than 25%, while the overall verification time is reduced by about 25%.

3 Loop Handling

One way to prove properties in model checking is by means of induction [1, 6, 8]. The k-induction method has already been successfully applied to verify hardware designs (represented as finite state machines) using a SAT solver, and first attempts to apply this technique to software have been made recently [5, 7]. We sketch the basic idea of our implementation in terms of temporal induction (i.e., the induction is carried out over the time steps of the finite state machines) [6, 7].

Our implementation uses an iterative deepening approach and checks, for each k up to a given maximum, three different cases called *base case*, *forward condition*, and *inductive step*. Intuitively, in the base case, we aim to find a counterexample with up to k loop unwindings; in the forward condition, we check that P holds in all states reachable within k unwindings; and in the inductive step, we check that whenever P holds for k unwindings, it also holds after the next unwinding of the system. We derive the verification conditions (VCs), which are denoted by $Base_k$, Fwd_k, and $Step_k$, respectively, for these program unwindings; if $Base_k$ is satisfiable, then we have found a violation of the safety property, and if Fwd_k or $Step_k$ are unsatisfiable, then the property holds.

The base case and the forward condition can be implemented with the right choice of existing command line parameters. For the base case we call ESBMC as follows:

```
esbmc --no-unwinding-assertions --unwind <i> <file>
```

This inserts an unwinding *assumption* consisting of the termination condition after each loop instead of the usual unwinding *assertion*. For the forward condition, we simply remove `--no-unwinding-assertions` from the call; note that we do not check whether paths are cycle-free as in [7]. The inductive step is more complex. In the approach by Grosse et al. [7], the state is havocked before the loop: all variables are assigned non-deterministic values. Then the loop is run $k - 1$ times, where all post-loop states are assumed to be different; in the loop body, all assertions are replaced by assumptions, which ensures that the chosen values satisfy a consequence of the (unknown) loop invariant. Lastly, the loop is run one final time, before the invariant is checked for the final state. However, as the competition benchmarks only check for reachability of the error label this schema does not have enough information to constrain the havocked variables. We thus havoc only the variables that occur in the loop's termination condition. This heuristic works well for the competition benchmarks.

4 Competition Approach

k-induction is more expensive than plain BMC because it uses iterative deepening and repeatedly unwinds the program, and because it produces more VCs (i.e., for base, forward, and step case). We thus combine k-induction and plain BMC; we first run the k-induction up to a maximum unwind bound, with an additional timeout to force early termination when its attempts fail, and then follow this by a plain BMC call:

```
esbmc --no-unwinding-assertions --partial-loops
      --unwind 6 <file>
```

`--partial-loops` removes the unwinding assumption, and thus allows paths where loops are executed only partially. We use a small script that glues together the ESBMC calls; it also sets the specific parameters for the memory safety category. The unwind bound and the distribution of the times allocated to both phases have been determined experimentally.

5 Results

With the k-induction enabled, ESBMC proves 1670 out of 1805 correct programs correct and finds errors in 424 of the 510 incorrect programs. However, it also claims errors in 17 correct programs and fails to find existing errors in 19 programs; most of these failures are in the `BitVectors` and `ControlFlowInteger` categories. ESBMC produces 115 time-outs, which are concentrated on the larger benchmarks (in particular `DeviceDrivers64` and `SystemC`). ESBMC produces good results for most categories except for `HeapManipulation`, where we opted out.

k-induction by itself is by far not as strong as plain BMC. In the training phase (which was run on similar hardware) it proved only 992 programs correct and found errors in only 98, although it also produced substantially fewer false results. The iterative deepening, with the repeated unwinding of the program, requires much more time for the symbolic execution of the program, and the higher number of VCs require more time in the SMT solver. However, in combination with plain BMC it is a useful technique, increasing the latter's raw score by about 200 marks. As expected, k-induction is particularly successful in the `Loops`-category, where it prevents 14 false results.

Availability. The script and self-contained binaries for 32-bit and 64-bit Linux environments are available at `www.esbmc.org`; versions for other operating systems are available on request. The competition version only uses the Z3 solver (V3.2).

Acknowledgements. The continued development of ESBMC is funded by the Royal Society and by INdT.

References

1. Bradley, A.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
2. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)
3. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans. Software Eng. 38(4), 957–974 (2012)
4. Cordeiro, L., Morse, J., Nicole, D., Fischer, B.: Context-Bounded Model Checking with ESBMC 1.17 (Competition Contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 534–537. Springer, Heidelberg (2012)
5. Donaldson, A., Kroening, D., Rümmer, P.: Automatic Analysis of Scratch-Pad Memory Code for Heterogeneous Multicore Processors. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 280–295. Springer, Heidelberg (2010)

6. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
7. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: MEMOCODE, pp. 113–122 (2010)
8. 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)

LLBMC: Improved Bounded Model Checking of C Programs Using LLVM[⋆] (Competition Contribution)

Stephan Falke, Florian Merz, and Carsten Sinz

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{stephan.falke,florian.merz,carsten.sinz}@kit.edu

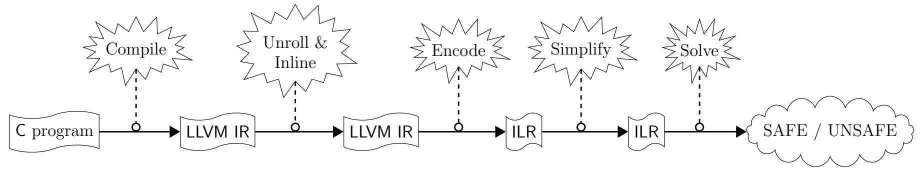
Abstract. LLBMC is a tool for detecting bugs and runtime errors in C and C++ programs. It is based on bounded model checking using an SMT solver and thus achieves bit-accurate precision. A distinguishing feature of LLBMC in contrast to other bounded model checking tools for C programs is that it operates on a compiler intermediate representation and not directly on the source code.

1 Verification Approach

Bounded model checking (BMC) of C, pioneered by Clarke, Kroening and Lerda [1], is a well-established method for detecting bugs and runtime errors. A number of mature tools for BMC of C programs already exists [1,2,6,8]. These tools only investigate finite paths in programs by bounding the number of loop iterations and the function call depth that is considered. This way, property checking becomes decidable using SMT solvers for the combined theory of bitvectors and arrays, where the latter are used to model the computer's main memory.

2 Software Architecture

Details on LLBMC's architecture and features can be found elsewhere [3,4,8,9]. The overall approach is summarized in the following figure:



First, the C program is compiled into the compiler intermediate representation LLVM IR [7]. Then, loops are unrolled, functions are inlined, and the resulting

⋆ This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

LLVM IR program is encoded into LLBMC's intermediate logic representation ILR. The ILR formula is finally simplified, lowered to an SMT formula, and solved using the SMT solver STP [5].

In comparison to the version that participated in SV-COMP 2012, this year's version of LLBMC offers the following improvements:

– Lazy, on-demand loop unrolling, function inlining, and ILR encoding.
– Uninitialized local variables are automatically set to nondeterministic values.
– SMT-based support for `memcpy`, `memset`, and `memmove` as an extension of the theory of arrays [4]. This is an alternative to providing C implementations.
– Extended support for many C library functions and `gcc` built-in functions. These library functions are provided as a module containing implementations of the functions in LLVM IR, where the module is automatically linked to the module obtained from the C program.
– Utilizes new versions of LLVM (version 3.1) and STP (revision 1668).

3 Strengths and Weaknesses of the Approach

LLBMC is tailored towards finding bugs in C programs, in particular memory-related ones. Detectable errors include common ones such as arithmetic overflow and underflow, invalid memory access operations, and invalid use of the memory allocation system (including invalid `free`s and memory leaks). Furthermore, LLBMC supports checking of user assertions and reachability of labels in the C program. In SV-COMP 2013, checking for most of these errors has been disabled and only reachability of the error label "ERROR" resp. only memory safety checks (in the category "MemorySafety") are performed.

In the competition, LLBMC is used with a maximal loop iteration bound of 10 and a maximal (recursive) function call depth of 2, where these bounds are increased iteratively based on the previous run of the tool. If no error is found within these maximal bounds, the instance is considered safe.

LLBMC did not participate in the categories "ControlFlowInteger-MemSimple" (LLBMC does not support the simplistic memory model), "Concurrency" (not supported by LLBMC), and "DeviceDrivers64" (since most of these programs implicitly also assume a simplistic memory model). In the categories where LLBMC participated, it performed very well, winning two categories ("BitVectors" and "Loops") and taking second place in four categories. LLBMC did not produce any incorrect result, but the time or memory limit was exhausted in 69 cases (out of the 1000 cases on which LLBMC was executed).

4 Tool Setup and Configuration

The version of LLBMC submitted to SV-COMP 2013 can be downloaded from

`http://llbmc.org/llbmc-sv-comp-13.zip`

LLBMC requires clang (version 3.1) in order to convert C input files to LLVM IR. The ZIP archive contains a wrapper shell script, llbmcc, to run LLBMC on individual C files that iteratively increases the loop iteration and function call depth if these bounds were reported to be insufficient by the previous run of LLBMC. In fact, the script runs LLBMC twice for each bound: In the first run it searches only for program errors, but does not check bounds. If no program error is found, a bounds check is performed in the second run.

By default, llbmcc only performs a reachability check for a basic block labelled "ERROR", but no other checks. In this case it outputs either SAFE, if the error label is unreachable (within the maximal bounds), or UNSAFE otherwise. Notice that LLBMC performs its analysis for a 32-bit machine and does not participate in the "DeviceDrivers64" category, which would require the analysis to be performed for a 64-bit machine (the script, however, supports -m64 as the first argument if the analysis for a 64-bit machine is desired). For the "MemorySafety" category, the script should be run with -mem-safety as the first argument. The script then checks for invalid frees, invalid memory dereferences, and memory leaks, but does not perform any other checks. In this case, it outputs either TRUE, if the program is memory safe (within the maximal bounds), or one of FALSE(p_valid-free), FALSE(p_valid-deref), or FALSE(p_valid-memtrack) if the corresponding memory safety property is violated in the verification task.

5 Software Project and Contributors

LLBMC is developed by Stephan Falke, Florian Merz, and Carsten Sinz at the Karlsruhe Institute of Technology (KIT) in Karlsruhe, Germany. The tool is available under either an unlimited non-commercial (academic) license or under an evaluation license that is valid for 30 days and suitable for a commercial setting. Further information on LLBMC can be found at http://llbmc.org.

References

1. 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)
2. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Proc. ASE 2009, pp. 137–148 (2009)
3. Falke, S., Merz, F., Sinz, C.: A theory of C-style memory allocation. In: Proc. SMT 2011, pp. 71–80 (2011)
4. Falke, S., Sinz, C., Merz, F.: A theory of arrays with set and copy operations (extended abstract). In: Proc. SMT 2012, pp. 97–106 (2012)
5. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
6. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. TCS 404(3), 256–274 (2008)

7. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO 2004, pp. 75–88 (2004)
8. Merz, F., Falke, S., Sinz, C.: `LLBMC`: Bounded Model Checking of `C` and `C++` Programs Using a Compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 146–161. Springer, Heidelberg (2012)
9. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: Proc. SSV 2010 (2010)

Predator: A Tool for Verification of Low-Level List Manipulation
(Competition Contribution)[*]

Kamil Dudka, Petr Müller, Petr Peringer, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Predator is a tool for automated formal verification of sequential C programs operating with pointers and linked lists. The core algorithms of Predator were originally inspired by works on separation logic with higher-order list predicates, but they are now purely graph-based and significantly extended to support various forms of low-level memory manipulation used in system-level code. This paper briefly introduces Predator and describes its participation in the Software Verification Competition SV-COMP'13 held at TACAS'13.

1 Predator Introduction

Predator is a tool for fully automated verification of sequential C programs with pointers and dynamic linked data structures, such as complex kinds of singly- and doubly-linked lists that can be circular, shared, and/or hierarchically nested in an arbitrary way. The long term goal of the Predator project is handling real system code, such as the Linux kernel. To achieve this, the tool strives to cope with implementation tricks and techniques used frequently by system programmers to obtain highly efficient code. Such techniques include pointer arithmetic, valid usage of pointers with invalid targets, operations with memory blocks, or reinterpretation of the memory contents. The degree to which Predator can deal with such techniques is currently to a large degree unique among fully automated shape analysis tools. Although Predator supports checking for error label reachability, it concentrates on an implicit detection of memory-related bugs. Hence, our main focus in SV-COMP'13 is the newly introduced *MemorySafety* competition category.

Predator is available in the form of a GCC plug-in, which brings several advantages. First, it is possible to re-use the existing build systems of GCC-based projects for running the verification without a need to manually process the source code. Predator, as a GCC plugin, can take advantage of the powerful parsing capabilities of GCC. Error messages are presented in a format compatible with GCC, hence Predator can be used with any IDE that can use GCC. Predator uses the low-level GIMPLE representation of the GCC intermediate code as an input for its analysis. By default, Predator disallows external function calls in order to exclude any side effects that could potentially

[*] This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (project MSM 0021630528), the BUT projects FIT-S-11-1 and FIT-S-12-1, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

break memory safety. The only allowed external functions are those which are properly modelled by Predator wrt. proving memory safety. Besides `malloc` and `free`, Predator supports selected memory manipulating functions like `memset`, `memcpy`, or `memmove`.

Predator is implemented in C++ and runs on Linux. The dependencies needed for building Predator are Boost, CMake, and the GCC plug-in development files. Predator is publicly available under the GPLv3 license.

2 Verification Approach

Predator was inspired by works on fully automated shape analysis using separation logic with higher-order inductive predicates [1]. However, Predator represents sets of heap configurations using a graph-based representation instead of separation logic formulae, which allows one to easily apply various efficient graph-based algorithms for dealing with the representation. Since SV-COMP'12, the graph-based representation has been redesigned into the form of the so-called *symbolic memory graphs* (SMGs) and made much more fine-grained (byte-precise) to allow for successfully verifying programs that use the above mentioned low-level memory manipulation techniques [3].

Predator iteratively computes sets of SMGs for each basic block of the CFG of the given program, covering all its reachable configurations. Termination of the analysis is aided by join and abstraction algorithms operating on SMGs. The join algorithm is based on simultaneously traversing two SMGs and merging their corresponding nodes. The abstraction uses the join algorithm to merge pairs of neighbouring nodes of the same SMG, together with their sub-SMGs, into a single list segment. Predator does not use any off-the-shelf decision procedure since an expensive conversion from our representation would be needed. Instead, entailment between SMGs is checked rather efficiently using the join algorithm, which is extended to compare on-the-fly the generality of the SMGs being joined. To allow for multiple views of a single block of memory, Predator implements *read* and *write reinterpretation* algorithms (needed, e.g., for dealing with unions and type-casts). For more details, see [3].

Predator can prove absence of common memory safety bugs, such as invalid dereferences or memory leaks. Apart from that, Predator uses the fact that SMGs make it possible to easily check whether a given pair of memory areas overlaps in order to check for bugs caused by memory overlapping in a way prohibited by the C language (as in the parameters of `memcpy`). Predator can provide diagnostic information accompanying errors or warnings, which due to the use of abstraction and join has a form of acyclic graphs covering multiple program paths possibly leading to the error.

Predator supports pointers with both positive and negative offsets from the beginning of allocated objects. Moreover, it even supports pointers with offsets given by integer intervals, which is needed to cope with some low-level code using, e.g., address alignment. Predator provides a simple support for integer data by tracking integers precisely up to some bound and then abstracting them to unknown values. Further details can be found in the tool paper [2] and in the technical report [3].

3 Benchmark Results

The latest release of Predator can be downloaded from its web page[1]. Specific instructions for building and running Predator within the SV-COMP'13 competition are located in the file `README-sv-comp-TACAS-2013` in the distribution of Predator.

Since the main focus of Predator is on memory- and pointer-related bugs, where it can utilize its precise analysis of reachable heap configurations, we concentrate on the *MemorySafety* and *HeapManipulation* categories. Compared to the SV-COMP'12 version of Predator, we successfully analysed many more test cases in the *MemorySafety* category. The new version of Predator managed all but one test case in this category. In particular, it did not scale well-enough to verify a program working with a 32KB array. On the other hand, even the new version of Predator still timed out on several tests in the *HeapManipulation* category. These test cases store integral data in list nodes in a way that prevents the list segment abstraction of Predator from applying. As Predator aims at verification of system software (including device drivers), we were interested in the *FeatureChecks* category as well. Predator successfully verified all test cases in the *ldv-regression* directory and a few test cases from the *ddv-machzwd* directory. Further, Predator achieved good results in the *ProductLines* category, where it successfully verified 585 of 597 test cases.

Results in the *SystemC*, *Loops*, and *ControlFlowInteger* categories had a higher ratio of false positives than in the above mentioned categories, but still with a majority of judgements being correct. The false positives are again caused mostly by a too coarse analysis of integers. For many cases in these categories, Predator was unable to provide an answer. The *BitVectors* category is problematic for Predator: safe test cases were often judged as unsafe because the byte-precise memory model used by Predator was too coarse for the bit-level operations. Due to undefined external functions, Predator was not able to analyze any test case from the *DeviceDrivers64* and *Concurrency* categories.

Over all categories, there was not a single case where Predator would issue an incorrect `TRUE` answer. This is a design goal of Predator, and it strengthens our claims that implementation of our verification techniques is sound.

Our future work includes handling of low-level tree data structures, a support for code fragment analysis, and better handling of integer data. Especially the last item would be beneficial for Predator's performance in some SV-COMP categories since we observed a high number of false positives caused by a too coarse analysis of integer data.

References

1. 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)
2. Dudka, K., Peringer, P., Vojnar, T.: Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
3. Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. Technical Report No. FIT-TR-2012-04, FIT BUT (2012)

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`

Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution[⋆] (Competition Contribution)

Jiri Slaby, Jan Strejček, and Marek Trtík

Faculty of Informatics, Masaryk University
Botanická 68a, 60200 Brno, Czech Republic
{slaby,strejcek,trtik}@fi.muni.cz

Abstract. Symbiotic is a tool for detection of bugs described by finite state machines in C programs. The tool combines three well-known techniques: instrumentation, program slicing, and symbolic execution. This paper briefly describes the approach of Symbiotic including its strengths, weaknesses, and modifications for SV-COMP 2013. Architecture and installation of the tool are described as well.

1 Verification Approach

Symbiotic implements our technique [4] that combines instrumentation, program slicing, and symbolic execution in order to detect bugs described by finite state machines. More precisely, we instrument a given program with code that tracks runs of state machines representing various erroneous behaviors. If an instrumented state machine enters an error location during a program execution, then the original program contains a bug specified by the machine. After instrumentation, we slice [5] the program to reduce its size without affecting runs of state machines. Finally, we symbolically execute [3] the sliced program to find bugs in the program.

As reachability of an `ERROR` label is the only bug considered in the SV-COMP, we have modified our instrumentor to put `assert(0)` function calls at `ERROR` labels in the code. Given the instrumented code, we execute Clang to produce an Llvm bitcode. This is in turn interprocedurally sliced with respect to slicing criteria, which are the instrumented `assert` calls. In other words we remove all the code except the one that has an effect on reachability of `assert` calls. The sliced Llvm bitcode is finally symbolically executed by Klee [1]. There are several possible outputs that Klee may generate. It can either find a reachable `assert` and report it, or finish the computation without any report, or terminate in some errant way (out of time, out of memory, invalid memory dereference, or some internal error for example). We map these to the demanded answers: `UNSAFE`, `SAFE`, or `UNKNOWN` respectively. This is taken care of in a simple scripted filter.

⋆ This work has been supported by The Czech Science Foundation (GAČR), grant No. P202/12/G061.

2 Software Architecture

Libraries/External Tools. For the code translation and for operations with the LLVM bitcode we use LLVM/CLANG 3.1[1]. The symbolic executor is KLEE which itself uses the STP constraint solver [2]. We obtained both KLEE and STP from the respective GIT repositories and used the snapshots. For more information about KLEE, see [1].

Software Structure and Architecture. The architecture described in Section 1 is summarized in Figure 1. The slicer is written as a plug-in for the optimizer `opt` from the LLVM suite. It is publicly available in a separate repository at `https://github.com/jirislaby/LLVMSlicer/`. The slicer improves the symbolic execution considerably. For ease of use, all parts of the tool pipeline are one by one run by a single script `runme`.

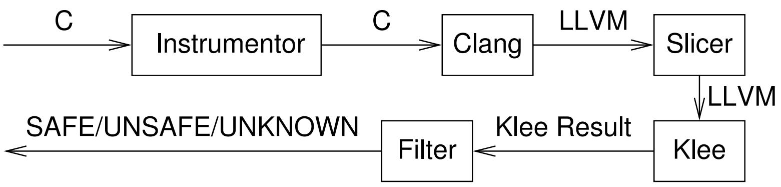


Fig. 1. Pipeline of the tool

Implementation Technology. The instrumentation is performed by a `bash` script using `sed`. The final filter also uses `bash` with the help of `grep`. The rest of the toolchain is written in C++ and compiled using `gcc`.

3 Discussion of Strengths and Weaknesses of the Approach

The strength of the tool lies in its high precision of the answers. In theory, the only source of incorrect answers is the slicer: it can completely remove an infinite loop in some cases and thus an unreachable `ERROR` label located below the loop may become reachable. However, there is no such case in the competition benchmarks.

All incorrect answers produced by our tool in the competition are due to bugs in implementation. Since the tool submission, we have fixed most of the bugs and improved the implementation a bit. The biggest change on the competition benchmarks can be seen in the category *FeatureChecks* (118 files):

[1] `http://llvm.org`

	Competition Version	Current Version
Correct Answers `SAFE/UNSAFE`	81	116
Incorrect Answers `SAFE/UNSAFE`	17	0
`UNKNOWN` (including timeouts)	20	2

In general, the main weakness of the tool is a high percentage of `UNKNOWN` results. These results come mainly from high computation cost of the symbolic execution of programs with loops or recursion. This problem is relieved by slicing, but there are still many cases where the sliced code remains complex and symbolic execution runs out of time or memory. Other sources of `UNKNOWN` results are internal errors of Klee and general limitations of constraint solving.

4 Tool Setup and Configuration

Download and Installation Instructions

- *Requirements:* `llvm-3.1`, `clang-3.1`.
- Download Symbiotic 1 at: `https://sf.net/projects/symbiotic/`.
- Change the current directory to `/opt` (this location is needed for Klee).
- Untar the archive with the tool.
- Change directory into `/opt/symbiotic`.
- Run `./runme <benchmark.c>` for each source file in the set.

Results Reported `SAFE/UNSAFE/UNKWNOWN` answers match the competition rules. The counterexample for an error in some `<benchmark.c>` is generated for each error path in the benchmark to `<benchmark.c>-klee-out/`. There is also a sliced code referred by all the error paths.

5 Software Project and Contributors

The concept of the tool has been developed by the authors of this paper. The tool was implemented by Jiri Slaby (contact person), Marek Trtík, and Ben Liblit (several fixes). It is available under the GNU GPLv2 License and is hosted by the Faculty of Informatics, Masaryk University.

References

1. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, pp. 209–224. USENIX Association (2008)
2. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
3. King, J.C.: Symbolic execution and program testing. Communications of ACM 19(7), 385–394 (1976)
4. Slabý, J., Strejček, J., Trtík, M.: Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 207–221. Springer, Heidelberg (2012)
5. Weiser, M.: Program slicing. In: Proceedings of ICSE, pp. 439–449. IEEE Press (1981)

Threader: A Verifier for Multi-threaded Programs (Competition Contribution)

Corneliu Popeea and Andrey Rybalchenko

Technische Universität München

Abstract. Threader is a tool that automates verification of safety and termination properties for multi-threaded C programs. The distinguishing feature of Threader is its use of reasoning that is compositional with regards to the thread structure of the verified program. This paper describes the verification approach taken by Threader and provides instructions on how to install and use the tool.

1 Verification Approach

Threader is a tool for verification of C programs based on predicate abstraction and refinement following the counterexample-guided abstraction refinement (CEGAR) paradigm [3]. There is a number of verification tools based on abstraction refinement that are successful for sequential programs [1,2,4,5,7,12]. This paper gives a brief description of specific features that were required to handle the concurrency benchmarks from the verification competition. Interested readers can find more details about the theory behind Threader in [6].

2 Software Architecture

Threader consists of two main components: a frontend for translating C programs in corresponding transition systems and a model checking back-end. The frontend is implemented in the OCaml language and relies on the CIL library [10]. Additional analyses are implemented in our frontend to handle the competition benchmarks (see next section for details). The model checker automates compositional reasoning of multi-threaded programs by implementing Owicki-Gries and rely-guarantee proof rules [9,11]. This model checker is implemented in the Prolog language and relies on the constraint solver for linear arithmetic CLP(Q) [8].

3 Discussion

In this section we present our experience in running Threader on the benchmarks from the Concurrency category.

Threader supports C programs with calls to Pthread library functions. To handle threads and mutex objects from the Pthread library, we require a pointer analysis that is more precise than the standard flow insensitive analysis available from the CIL library. As a solution to this problem, we implemented a context-sensitive pointer analysis that is explicit about some heap allocated objects and sound for multi-threaded programs.

Creation of threads in loops is another difficulty for Threader, since our model checker assumes a finite number of threads during verification. To handle this problem, we implemented a frontend analysis to compute the number of loop iterations and consequently the number of threads to be created. For all the competition benchmarks, this analysis is precise and we obtain constant values for the number of threads. As future work we would like to handle cases where the number of threads cannot be precisely computed statically, i.e., to be able to do automatic verification of parameterized systems.

Another difficulty for automatic verifiers is the analysis of array objects. Here Threader takes a pragmatic approach automating verification for some particular universal properties over the elements of an array. This reasoning is sufficient to handle three benchmarks (`indexer_safe.i`, `stack_unsafe.i` and `stack_safe.i`). Precise results for the four queue benchmarks require invariants that relate contents of different array objects and cannot be currently handled by Threader.

The set of Concurrency benchmarks contains some benchmarks that are pre-processed using the Simplify CIL module (the `*.cil.c` benchmarks). These benchmarks are presented as three-address-code with a significant number of temporary variables, with 'for' statements transformed into loops with 'goto' statements indicating the loop exit, and with array operations expressed using pointer arithmetic. Threader benefits from the CIL framework that allows an easy recovery of the high-level information regarding loops and array operations. Therefore we observed (almost) identical verification results and times for both the `*.cil.c` and the `*.i` forms of the benchmarks.

In general our verifier is designed not to miss bugs present in the C programs. We list here some of the significant advantages of Threader that facilitate a sound analysis of multi-threaded programs.

- Threader is applicable to arbitrary (or ad-hoc) synchronization patterns, not only nested locking patterns or datarace free code.
- Threader does not restrict the analysis to a bounded number of context-switches, but instead deals with an unbounded number of context switches.
- Threader is not restricted to programs with thread-modular proofs and can handle the general case of non-thread-modular proofs required for example by the Fibonacci competition benchmarks.

To summarize, we ran Threader on the 32 benchmarks from the Concurrency category and obtained a total of 43 out of the 49 points available in this category. Threader reports SAFE and UNSAFE correctly for 28 benchmarks. For the other four benchmarks (`queue_unsafe.cil.c`, `queue_unsafe.i`, `queue_ok_safe.cil.c`, `queue_ok_safe.i`), THREADER returns UNKNOWN due to limitations in handling quantified array invariants. (We are not aware of any automatic verification tool that can handle these benchmarks.) A SAFE result leads to the creation of an abstract reachability tree that represents a correctness proof (see generated file `art.dot`). An UNSAFE result leads to the creation of a counterexample in dotty format (see generated file `cex.dot`).

4 Tool Setup

THREADER can be downloaded from `http://www7.in.tum.de/tools/threader/`.

THREADER is provided as a set of statically compiled binaries for the Linux x86-64 architecture. A script is provided to invoke THREADER with predefined options for the competition. The tool should be run as follows: `./threader.sh <file.c>`. The working directory (`PWD`) must be the directory where THREADER's files are located.

Acknowledgements. We gratefully acknowledge the help of Ashutosh Gupta on designing and implementing various aspects of the previous version of THREADER. This research was supported in part by ERC project 308125 VeriSynth.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL (2002)
2. 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)
3. 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)
4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
5. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A Software Verifier Based on Horn Clauses (Competition Contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
6. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344 (2011)
7. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
8. Holzbaur, C.: OFAI clp(q,r) Manual, Edition 1.3.3. Austrian Research Institute for Artificial Intelligence, Vienna (1995), TR-95-09

9. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
10. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
11. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. 6, 319–340 (1976)
12. 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 (2007)

UFO: Verification with Interpolants and Abstract Interpretation
(Competition Contribution)

Aws Albarghouthi[1], Arie Gurfinkel[2], Yi Li[1],
Sagar Chaki[2], and Marsha Chechik[1]

[1] Department of Computer Science, University of Toronto, Canada
[2] Software Engineering Institute, Carnegie Mellon University, USA

1 Verification Approach

The algorithms underlying Ufo are described in [1–3]. The Ufo tool is described in more detail in [4].

Ufo marries the power and efficiency of numerical Abstract Interpretation (AI) domains [6] with the generalizing ability of interpolation-based software verification in an abstraction refinement loop. More formally: given a program P, a safety property φ, and some abstract domain $\mathcal{A}$, Ufo starts by computing an inductive invariant $\mathcal{I}$ of P in $\mathcal{A}$. If $\mathcal{I} \Rightarrow \varphi$, then we know that P satisfies φ, i.e., P cannot reach any of the error states characterized by $\neg\varphi$. Otherwise, if $\mathcal{I} \not\Rightarrow \varphi$, Ufo uses SMT solving to check whether the alarm raised by $\mathcal{I}$ maps to a real bug in the code. To do so, Ufo encodes all of the program paths explored by abstract interpretation as a formula, and uses an SMT solver to check its satisfiability. If the formula is satisfiable, an erroneous execution is reported to the user. Otherwise, an interpolation technique guided by the results of AI is used to strengthen $\mathcal{I}$ into $\mathcal{I}'$, where $\mathcal{I}' \Rightarrow \varphi$. If $\mathcal{I}'$ is no longer inductive, abstract interpretation continues from the set of states described by $\mathcal{I}'$. Otherwise, the program is safe.

2 Software Architecture

Ufo is implemented in C++ in the LLVM compiler infrastructure [7] as a general verification framework. Its architecture is shown in Fig. 1. In what follows, we describe our instantiation of the framework for the purposes of the competition.

Preprocessing Phase. The first step in this phase is converting a given program into the LLVM intermediate representation. Following that, we perform compiler optimizations and preprocessing in order to simplify the verification process. As a preprocessing step, we initialize uninitialized variables using non-deterministic functions. This is used to bridge the gap between the verification semantics (which assume a non-determinsitic assignment) and compiler semantics, which presets unitialized variables with the goal of optimizing the code. For optimizations, we perform a number of program simplifications such as function

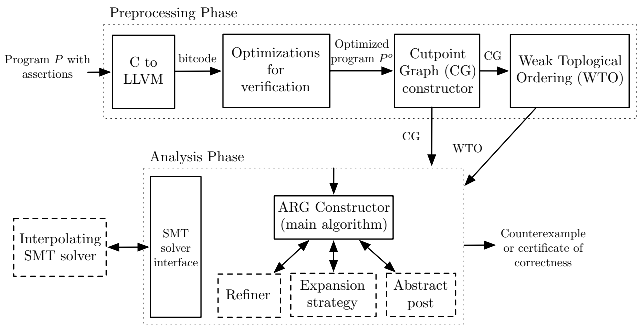

Preprocessing Phase
Program P with assertions
C to LLVM
bitcode
Optimizations for verification
Optimized program P^o
Cutpoint Graph (CG) constructor
CG
Weak Toplogical Ordering (WTO)
CG
WTO
Analysis Phase
Interpolating SMT solver
SMT solver interface
ARG Constructor (main algorithm)
Refiner
Expansion strategy
Abstract post
Counterexample or certificate of correctness

Fig. 1. The architecture of UFO [4]

inlining, converting the program into the static single assignment (SSA) form by reducing memory operations into SSA registers, removing dead code, etc.

After the optimization step, we represent the program as a Cutpoint Graph (CG), a control-flow graph where each node is a cutpoint in the original program and each edge is a loop-free execution between two cutpoints. Then, a Weak Topological Ordering (WTO) [5] is computed for the CG and used later as the abstract interpretation strategy.

Analysis Phase. The main algorithm constructs an Abtract Reachability Graph (ARG), a labelled unrolling of the program that represents an inductive invariant using a given abstract domain. The ARG contructor is parameterized by the abstract domain used and the refinement strategy:

– *Abstract domains*: The abstract domains we use are Box (intervals), Boxes [6] (intervals with disjunctions), and Cartesian and Boolean predicate abstraction. Our experiments have shown that different domains are useful for different problems and there is no clear winner. Thus, for the purposes of the competition, we instrumented Ufo to run multiple analysis instances with different domains in parallel, reporting the results of the fastest instance.
– *Refinement*: As a refinement strategy, we used AI-guided DAG interpolants from [1]. DAG interpolants annotate a directed acyclic graph of paths using a single call to an SMT solver, delegating the process of path enumeration to the SMT solver. In comparison, other techniques, e.g., Impact [8], unroll the program into a tree, potentially having to refine exponentially many paths in the size of the program. Furthermore, our refinement strategy uses the invariant computed by abstract interpretation in the encoding, often resulting in weaker interpolants and faster SMT solving time.

The Z3 [9] SMT solver is used for satisfiability checking, and MathSAT5[1] for computing interpolants. Due to the efficiency of Z3, we use it to shrink an interpolation query by computing an UNSAT core of a formula before handing it to MathSAT5 for satisfiability checking and interpolation.

[1] mathsat.fbk.eu

The analysis phase results in either `SAFE` – a safe inductive invariant has been computed, `CEX` – a counterexample has been found, or `UNKNOWN` – implying that UFO failed to produce a conclusive result. Counterexamples are produced as traces over basic blocks in the LLVM intermediate representation of the program.

3 Strengths and Weaknesses

UFO has been succesfully applied to `ControlFlowIntegers`, `SystemC`, `DeviceDrivers64`, and `ProductLines`. Currently, UFO uses linear arithmetic to model semantics of sequential C programs, making it imprecise for categories such as `BitVectors` (requiring bit-level precision), `HeapManipulation` (requiring heap tracking), and `Concurrency` (requiring thread handling). Another weakness is UFO's reliance on multiple tools for the front-end: LLVM 2.6, LLVM 2.9, and CIL. This increases the trusted computing base and makes it harder to maintain.

The power of UFO lies in its parameterized nature, allowing instantiations with different abstract domains and providing a general framework for experimenting with verification algorithms.

Tool Setup and Configuration. UFO is available for download from `bitbucket.org/arieg/ufo/wiki/svcomp13.wiki`. The options for running the tool are:

```
./bin/ufo-svcomp-par.py [-m64] --cex=FILE input
```

where `-m64` turns on 64-bit model, `--cex` is the location of the counter-example, and `input` is a C file.

Acknowledgment. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a FFRDC[2]. We also acknowledge financial support provided by NSERC, NECSIS and Alexander Graham Bell Scholarship.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig Interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012)

[2] NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. This material has been approved for public release and unlimited distribution. (DM-0000076)

2. 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)
3. 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)
4. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 672–678. Springer, Heidelberg (2012)
5. Bourdoncle, F.: Efficient Chaotic Iteration Strategies with Widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
6. Gurfinkel, A., Chaki, S.: Boxes: A Symbolic Abstract Domain of Boxes. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 287–303. Springer, Heidelberg (2010)
7. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004, pp. 75–88 (2004)
8. 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)
9. 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)

Ultimate Automizer with SMTInterpol[⋆]
(Competition Contribution)

Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski

University of Freiburg, Germany

Abstract. Ultimate Automizer is an automatic software verification tool for C programs. This tool is the first implementation of *trace abstraction*, which is an automata-theoretic approach to software verification. The implemented algorithm uses *nested interpolants* in its interprocedural program analysis. The interpolating SMT solver SMTInterpol is used to compute Craig interpolants.

1 Verification Approach

UltimateAutomizer verifies a C program by first executing several program transformations and then performing an interpolation based variant of trace abstraction [5].

As a first step we translate the C program into a Boogie [7] program. Next, the Boogie program is translated into an interprocedural control flow graph [8]. As an optimization we do not label the edges with single program statements but with loop free code blocks of the program [2].

In our algorithm, the program is represented by an automaton which accepts all error traces of the program. An error trace is a labeling of an initial path in the control flow graph which leads to the error location. If all error traces are infeasible with respect to the semantics of the programming language, the program is correct.

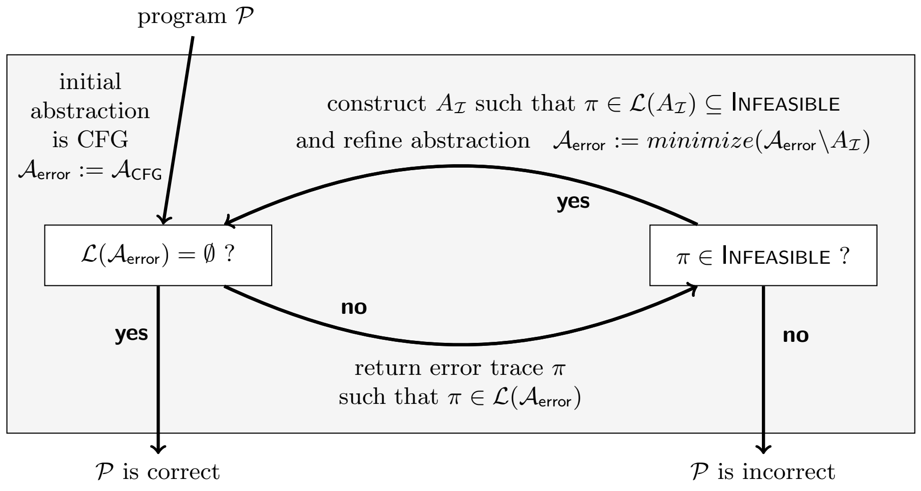
We use the CEGAR algorithm depicted below. Our abstraction is a nested word automaton [1] $\mathcal{A}_{\mathsf{error}}$ which accepts only error traces of the program. We iteratively subtract from our abstraction $\mathcal{A}_{\mathsf{error}}$ the language of an automaton $\mathcal{A}_{\mathcal{I}}$ which accepts only infeasible traces. The algorithm terminates if the language of $\mathcal{A}_{\mathsf{error}}$ is empty or a feasible error trace π was found.

The automaton $\mathcal{A}_{\mathcal{I}}$ which accepts only infeasible traces is constructed as follows. If the error trace π is infeasible we consider π as a straight-line program which has a Hoare annotation $\mathcal{I}$ where the initial location is labeled with the *true* predicate and the final state assertion is the *false* predicate. We call such

⋆ This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR14 AVACS).

a Hoare annotation $\mathcal{I}$ a sequence of nested interpolants [6] for π. We compute a sequence of nested interpolants by recursively computing sequences of Craig interpolants.

After subtracting the language of the interpolant automaton $\mathcal{A}_{\mathcal{I}}$ from our abstraction $\mathcal{A}_{\mathsf{error}}$, we apply a minimization for nested word automata which preserves the language of $\mathcal{A}_{\mathsf{error}}$, but reduces the number of states significantly in many cases.


program $\mathcal{P}$
initial abstraction is CFG $\mathcal{A}_{\mathsf{error}} := \mathcal{A}_{\mathsf{CFG}}$
construct $A_{\mathcal{I}}$ such that $\pi \in \mathcal{L}(A_{\mathcal{I}}) \subseteq$ INFEASIBLE
and refine abstraction $\mathcal{A}_{\mathsf{error}} := minimize(\mathcal{A}_{\mathsf{error}} \setminus A_{\mathcal{I}})$
yes
$\mathcal{L}(\mathcal{A}_{\mathsf{error}}) = \emptyset$?
$\pi \in$ INFEASIBLE ?
no
yes
no
return error trace π such that $\pi \in \mathcal{L}(\mathcal{A}_{\mathsf{error}})$
$\mathcal{P}$ is correct
$\mathcal{P}$ is incorrect

2 Software Architecture

Ultimate Automizer is one toolchain of the software analysis framework Ultimate[1] which is implemented in Java. Ultimate offers data structures for different representations of a program, plugins which analyze or transform a program, and an interface for the communication with SMT-LIBv2 compatible theorem provers. For parsing C programs, we use the C parser of the Eclipse CDT project[2]. The operations on nested word automata are implemented in the Ultimate Automata Library. As interpolating SMT solver we use SMTInterpol[3] [3].

3 Discussion of Approach

Conceptually, our approach is applicable to each class of programs whose semantics can be defined via SMT formulas. However, the current implementation of Ultimate Automizer supports only sequential programs and does neither support arrays, pointers, nor bitvector operations.

[1] http://ultimate.informatik.uni-freiburg.de/
[2] http://www.eclipse.org/cdt/
[3] http://ultimate.informatik.uni-freiburg.de/smtinterpol/

4 Tool Setup and Configuration

Our competition candidate is a version of Ultimate with a command line user interface that contains a version of SMTInterpol and can be downloaded from the following website:

`http://ultimate.informatik.uni-freiburg.de/automizer`

The zip archive in which Ultimate Automizer is shipped contains the bash script `automizerSV-COMP.sh` which calls Ultimate with all parameters that are necessary to verify C programs using the Ultimate Automizer toolchain. In order to verify the C program `fnord.c`, use the directory where you extracted the zip archive as your working directory and execute the following command:

```
automizerSV-COMP.sh fnord.c
```

5 Software Project and Contributors

Our software analysis framework Ultimate was started as a bachelor thesis [4]. In the last years, many students contributed plugins or improved the framework itself. Soon we will release two user interfaces for Ultimate, a web interface and a plugin for the Eclipse CDT project. In both user interfaces you can also use the UltimateAutomizer toolchain to verify C programs.

The Authors thank Alex Saukh and Stefan Wissert for their contributions to the plugin which translates C programs to Boogie programs. Furthermore the Authors thank all developers that contributed to Ultimate, to the Ultimate Automata Library, or to SMTInterpol.

References

1. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM 56(3), 16:1–16:43 (2009)
2. 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)
3. Christ, J., Hoenicke, J., Nutz, A.: Proof Tree Preserving Interpolation. In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 123–137. Springer, Heidelberg (2013)
4. Dietsch, D.: STALIN: A plugin-based modular framework for program analysis. Bachelor Thesis, Albert-Ludwigs-Universität, Freiburg, Germany (2008)
5. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of Trace Abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009)
6. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 471–482. ACM (2010)
7. Leino, K.R.M.: This is Boogie 2. Manuscript working draft, Microsoft Research, Redmond, WA, USA (June 2008), `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`
8. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995, pp. 49–61. ACM (1995)

Author Index