

ARCoSS

LNCS 12079

Armin Biere
David Parker (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

26th International Conference, TACAS 2020
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2020
Dublin, Ireland, April 25–30, 2020, Proceedings, Part II

2
Part II



 Springer Open

Founding Editors

Gerhard Goos, Germany
Juris Hartmanis, USA

Editorial Board Members

Elisa Bertino, USA
Wen Gao, China
Bernhard Steffen , Germany

Gerhard Woeginger , Germany
Moti Yung, USA

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome 'La Sapienza', Italy*
Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *TU Munich, Germany*
Benjamin C. Pierce, *University of Pennsylvania, USA*
Bernhard Steffen , *University of Dortmund, Germany*
Deng Xiaotie, *Peking University, Beijing, China*
Jeannette M. Wing, *Microsoft Research, Redmond, WA, USA*

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

Armin Biere · David Parker (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

26th International Conference, TACAS 2020
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2020
Dublin, Ireland, April 25–30, 2020
Proceedings, Part II

Editors

Armin Biere 
Johannes Kepler University
Linz, Austria

David Parker 
University of Birmingham
Birmingham, UK



ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-45236-0 ISBN 978-3-030-45237-7 (eBook)
<https://doi.org/10.1007/978-3-030-45237-7>

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© The Editor(s) (if applicable) and The Author(s) 2020. This book is an open access publication.

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

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

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

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

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

ETAPS Foreword

Welcome to the 23rd ETAPS! ETAPS 2020 was originally planned to take place in Ireland in its beautiful capital Dublin. Because of the Covid-19 pandemic, this was changed to an online event on July 2, 2020.

ETAPS 2020 is the 23rd instance of the European Joint Conferences on Theory and Practice of Software.

ETAPS is an annual federated conference established in 1998, and consists of four conferences: ESOP, FASE, FoSSaCS, and TACAS.

Each conference has its own Program Committee (PC) and its own Steering Committee (SC).

The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations of programming language developments, analysis tools, and formal approaches to software engineering.

Organizing these conferences in a coherent, highly synchronized conference programme, enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences.

On the weekend before the main conference, numerous satellite workshops take place that attract many researchers from all over the globe. Also, for the second time, an ETAPS Mentoring Workshop is organized.

This workshop is intended to help students early in the program with advice on research, career, and life in the fields of computing that are covered by the ETAPS conference.

ETAPS 2020 received 424 submissions in total, 129 of which were accepted, yielding an overall acceptance rate of 30.4%.

I thank all the authors for their interest in ETAPS, all the reviewers for their reviewing efforts, the PC members for their contributions, and in particular the PC (co-) chairs for their hard work in running this entire intensive process.

Last but not least, my congratulations to all authors of the accepted papers!

Because of the change to an online event, most of the original ETAPS program had to be cancelled. The ETAPS afternoon featured presentations of the three best paper awards, the Test-of-Time award and the ETAPS PhD award. The invited and tutorial speakers of ETAPS 2020 will be invited for ETAPS 2021, and all authors of accepted ETAPS 2020 papers will have the opportunity to present their work at ETAPS 2021.

ETAPS 2020 originally was supposed to place in Dublin, Ireland, organized by the University of Limerick and Lero. The local organization team consisted of Tiziana Margaria (UL and Lero, general chair), Vasileios Koutavas (Lero@UCD), Anila Mjeda (Lero@UL), Anthony Ventresque (Lero@UCD), and Petros Stratis (Easy Conferences). I would like to thank Tiziana and her team for all the preparations, and we hope there will be a next opportunity to host ETAPS in Dublin.

ETAPS 2020 is further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science),

EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology).

The ETAPS Steering Committee consists of an Executive Board, and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST.

The Executive Board consists of Holger Hermanns (Saarbrücken), Marieke Huisman (Twente, chair), Joost-Pieter Katoen (Aachen and Twente), Jan Kofron (Prague), Gerald Lüttgen (Bamberg), Tarmo Uustalu (Reykjavik and Tallinn), Caterina Urban (INRIA), and Lenore Zuck (Chicago).

Other members of the steering committee are:

Armin Biere (Linz)
Jordi Cabot (Barcelona)
Jean Goubault-Larrecq (Cachan)
Jan-Friso Groote (Eindhoven)
Esther Guerra (Madrid)
Jurriaan Hage (Utrecht)
Reiko Heckel (Leicester)
Panagiotis Katsaros (Thessaloniki)
Stefan Kiefer (Oxford)
Barbara König (Duisburg)
Fabrice Kordon (Paris)
Jan Kretinsky (Munich)
Kim G. Larsen (Aalborg)
Tiziana Margaria (Limerick)
Peter Müller (Zurich)
Catuscia Palamidessi (Palaiseau)
Dave Parker (Birmingham)
Andrew M. Pitts (Cambridge)
Peter Ryan (Luxembourg)
Don Sannella (Edinburgh)
Bernhard Steffen (Dortmund)
Mariëlle Stoelinga (Twente)
Gabriele Taentzer (Marburg)
Christine Tasson (Paris)
Peter Thiemann (Freiburg)
Jan Vitek (Prague)
Heike Wehrheim (Paderborn)
Anton Wijs (Eindhoven), and
Nobuko Yoshida (London)

I'd like to take this opportunity to thank all authors, attendants, organizers of the satellite workshops, and Springer-Verlag GmbH for their support.

I hope you all enjoyed the ETAPS 2020 afternoon.

July 2020

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

TACAS 2020 was the 26th edition of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems conference series. TACAS 2020 was part of the 23rd European Joint Conferences on Theory and Practice of Software (ETAPS 2020).

TACAS is a forum for researchers, developers, and users interested in rigorously based tools and algorithms for the construction and analysis of systems. The conference aims to bridge the gaps between different communities with this common interest and to support them in their quest to improve the utility, reliability, flexibility, and efficiency of tools and algorithms for building systems. TACAS solicited four types of submissions:

- *Research papers* advancing the theoretical foundations for the construction and analysis of systems
- *Case study papers* with an emphasis on a real-world setting
- *Regular tool papers* presenting a new tool, a new tool component, or novel extensions to an existing tool and requiring an artifact submission
- *Tool demonstration papers* focusing on the usage aspects of tools, also subject to the artifact submission requirement

This year 155 papers were submitted to TACAS, consisting of 111 research papers, 8 case study papers, 19 regular tool papers, and 17 tool demo papers. Individual authors were limited to a maximum of three submissions. Each paper was reviewed by at least three Program Committee (PC) members, who also provided feedback whether certain papers should go through a rebuttal process.

The chairs asked for 59 rebuttals, usually following such rebuttal recommendations by PC members. In parallel to PC reviewing, the Artifact Evaluation Committee (AEC) reviewed the artifacts. A formal summary review of this evaluation was made available to the PC members and taken into account in the discussion phase. The case study chair and the tools chair made sure that identical reviewing and selection criteria were applied within their respective class of papers. After this thorough reviewing, rebuttal and discussion phase, a total of 48 papers were accepted, including 31 research papers, 4 case study papers, 5 regular tool papers and 8 tool demo papers.

As in 2019, TACAS 2020 included an artifact evaluation (AE) for all types of papers. There were two rounds of the AE: for regular tool papers and tool demonstration papers AE was compulsory and artifacts had to be submitted to the first round. For research and case study papers, it was voluntary, and artifacts could be submitted to either the first or the second round. The results of the first round were communicated to

the TACAS PC before their discussion phase so that the quality of the artifact could be considered prior to the TACAS decision making. Each artifact was evaluated independently by at least three reviewers. All accepted papers with accepted artifacts received a badge which is added to the title page of the respective paper if desired by the authors.

The AEC used a two-phase reviewing process: reviewers first performed an initial check to see whether the artifact was technically usable and whether the accompanying instructions were consistent, followed by a full evaluation of the artifact. The main criteria for artifact acceptance was consistency with the paper, with completeness, and documentation being handled in a more lenient manner as long as the artifact was useful overall.

In the first round, out of 44 artifact submissions, 29 were accepted and 15 were rejected. This corresponds to an acceptance rate of 66%. Out of the 36 artifacts for regular tool papers and tool demonstration papers, 25 artifacts were accepted and 11 artifacts were rejected resulting in an acceptance rate of 69%. In all but five cases, tool papers whose artifacts did not pass the evaluation were rejected. Those 5 artifacts were invited for submission in the second evaluation round and 3 of these artifacts were resubmitted and successfully evaluated. Overall, out of the 20 artifacts submitted to the second evaluation round, 17 were accepted and 3 were rejected resulting in an acceptance rate of 85%.

TACAS 2020 also hosted the 9th International Competition on Software Verification (SV-COMP 2020), chaired and organized by Dirk Beyer. The competition had again a high participation: 28 verification systems with developers from 11 countries were submitted for the systematic comparative evaluation, including 3 submissions from industry. Six teams contributed validators for verification witnesses. The TACAS proceedings includes the competition report and short papers describing 11 of the participating verification systems. These papers were reviewed by a separate SV-COMP program committee; each of the papers was assessed by at least three reviewers. Two sessions in the TACAS program were reserved for the presentation of the results: the summary by the SV-COMP chair and the participating tools by the developer teams in the first session, and the open community meeting in the second session.

We are grateful to everyone who helped to make TACAS 2020 a success. In particular, we would like to thank all PC members, external reviewers, and the members of the AEC for their detailed and informed reviews and for their discussions during the virtual PC and AEC meetings. The collection and selection of papers was organized through the EasyChair Conference System and the proceedings volumes were published with the help of Springer; we thank them all for their assistance. We

also thank the SC for their advice, the Organizing Committee of ETAPS 2020 and its general chair (Tiziana Margaria) and the chair of the ETAPS Executive Board (Marieke Huisman).

March 2020

Armin Biere
David Parker
PC Chairs
Marijn Heule
Case Study Chair
Falk Howar
Tools Chair
Dirk Beyer
Competition Chair
Arnd Hartmanns
Martina Seidl
AEC Chairs

Organization

Program Committee

Christel Baier	TU Dresden, Germany
Ezio Bartocci	Vienna University of Technology, Austria
Dirk Beyer	LMU Munich, Germany
Armin Biere (Chair)	Johannes Kepler University Linz
Jasmin Blanchette	Vrije Universiteit Amsterdam, The Netherlands
Roderick Bloem	TU Graz, Austria
Hana Chockler	King's College London, UK
Alessandro Cimatti	FBK-irst, Italy
Rance Cleaveland	University of Maryland, USA
Goran Frehse	Université Grenoble Alpes, France
Martin Fränzle	Carl von Ossietzky Univ. Oldenburg, Germany
Orna Grumberg	Technion - Israel Institute of Technology
Kim Guldstrand Larsen	Aalborg University, Denmark
Holger Hermanns	Universität des Saarlandes, Germany
Marijn Heule	Carnegie Mellon University, USA
Falk Howar	TU Clausthal, IPSSE, Germany
Benjamin Kiesl	CISPA Helmholtz Center for Inf. Security, Germany
Laura Kovacs	Vienna University of Technology, Austria
Jan Kretinsky	TU Munich, Germany
Wenchao Li	Boston University, USA
Ken McMillan	Microsoft, USA
Aina Niemetz	Stanford University, USA
Gethin Norman	University of Glasgow, UK
David Parker (Chair)	University of Birmingham, UK
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Nir Piterman	University of Gothenburg, Sweden
Kristin Yvonne Rozier	Iowa State University, USA
Philipp Ruemmer	Uppsala University, Sweden
Natasha Sharygina	Università della Svizzera italiana, Switzerland
Bernhard Steffen	TU Dortmund, Germany
Jan Strejček	Masaryk University, Czech Republic
Michael Tautschnig	Queen Mary University of London, UK
Jaco van de Pol	Aarhus University, Denmark
Tom van Dijk	University of Twente, The Netherlands
Christoph Wintersteiger	Microsoft, UK

Artifact Evaluation Committee

Pranav Ashok	TU Munich, Germany
Peter Backeman	Uppsala University, Sweden
Ismail Lahkim Bennani	Inria, France
Carlos E. Budde	University of Twente, The Netherlands
Karlheinz Friedberger	LMU Munich, Germany
Arnd Hartmanns (Chair)	University of Twente, The Netherlands
Jannik Hüls	Westfälische Wilhelms-Univ. Münster, Germany
Ahmed Irfan	Stanford University, USA
Martin Jonas	Masaryk University, Czech Republic
William Kavanagh	University of Glasgow, UK
Brian Kempa	Iowa State University, USA
Michaela Klauk	Universität des Saarlandes, Germany
Sascha Klüppelholz	TU Dresden, Germany
Bettina Könighofer	TU Graz, Austria
Sophie Lathouwers	University of Twente, The Netherlands
Florian Lonsing	Stanford University, USA
Juraj Major	Masaryk University, Czech Republic
Tobias Meggendorfer	TU Munich, Germany
Vince Molnar	Budapest Univ. of Tech. and Economics, Hungary
Alnis Murtovi	TU Dortmund, Germany
Chris Novakovic	University of Birmingham, UK
Nicola Paoletti	Royal Holloway University of London, UK
Tim Quatmann	RWTH Aachen University, Germany
Martina Seidl (Chair)	Johannes Kepler University Linz, Austria
Leander Tentrup	Universität des Saarlandes, Germany
Freak van der Berg	University of Twente, The Netherlands
Marcell Vazquez-Chanlatte	University of California at Berkeley, USA
Matthias Volk	RWTH Aachen University, Germany
Petar Vukmirovic	Vrije Universiteit Amsterdam, The Netherlands
Maximilian Weininger	TU Munich, Germany

SV-COMP – Program Committee and Jury

Dirk Beyer (Chair)	LMU Munich, Germany
Viktor Malík (2LS)	BUT Brno, Czech Republic
Lei Bu (BRICK)	Nanjing University, China
Michael Tautschnig (CBMC)	Amazon Web Services, UK
Willem Visser (COASTAL)	Stellenbosch University, South Africa
Vadim Mutilin (CPA-BAM-BnB)	ISP RAS, Russia
Martin Spiessl (CPA-Seq)	LMU Munich, Germany
Pavel Andrianov (CPALockator)	ISP RAS, Russia

Hernán Ponce de León (Dartagnan)	Bundeswehr University Munich, Germany
Henrich Lauko (DIVINE)	Masaryk University, Czechia
Felipe R. Monteiro (ESBMC)	Fed. Univ. of Amazonas, Brazil
Benjamin Quiring (GACAL)	Northeastern University, USA
Vaibhav Sharma (Java-Ranger)	University of Minnesota, USA
Philipp Rueemmer (JayHorn)	Uppsala University, Sweden
Peter Schrammel (JBMC)	University of Sussex, UK
Falk Howar (JDart)	TU Dortmund, Germany
Omar Inverso (Lazy-CSeq)	Gran Sasso Science Institute, Italy
Herbert Rocha (Map2Check)	Universidade Federal do Amazonas, Brazil
Philipp Berger (NITWIT)	RWTH Aachen, Germany
Cedric Richter (PeSCo)	Paderborn University, Germany
Saurabh Joshi (Pinaka)	IIT Hyderabad, India
Veronika Šoková (PredatorHP)	BUT Brno, Czech Republic
Willem Visser (SPF)	Amazon Web Services, USA
Marek Chalupa (Symbiotic)	Masaryk University, Czech Republic
Matthias Heizmann (UAutomizer)	University of Freiburg, Germany
Alexander Nutz (UKojak)	University of Freiburg, Germany
Daniel Dietsch (UTaipan)	University of Freiburg, Germany
Priyanka Darke (VeriAbs)	Tata Consultancy Services, India
Raveendra K. Medicherla (VeriFuzz)	Tata Consultancy Services, India
Liangze Yin (Yogar-CBMC)	Nat. Univ. of Defense Technology, China

Steering Committee

Bernhard Steffen (Chair)	TU Dortmund, Germany
Dirk Beyer	LMU Munich, Germany
Rance Cleaveland	University of Maryland, USA
Holger Hermanns	Universität des Saarlandes, Germany
Kim G. Larsen	Aalborg University, Denmark

Additional Reviewers

Alexandre Dit Sandretto, Julien
Asadi, Sepideh
Ashok, Pranav
Avigad, Jeremy
Baanen, Tim
Bacci, Giorgio
Bacci, Giovanni
Backeman, Peter
Bae, Kyungmin
Barbosa, Haniel
Bentkamp, Alexander
Berani Abdelwahab, Erzana
Biewer, Sebastian
Blahoudek, Fanda
Blich, Martin
Bozga, Marius
Bozzano, Marco
Bønneland, Frederik M.
Cerna, David
Ceska, Milan
Chalupa, Marek
Chapoutot, Alexandre
Dierl, Simon
Dureja, Rohit
Ebrahimi, Masoud
Eisentraut, Julia
Endrullis, Jörg
Ernst, Gidon
Esen, Zafer
Fan, Jiameng
Fazekas, Katalin
Fedyukovich, Grigory
Fleury, Mathias
Fokkink, Wan
Forets, Marcelo
Freiberger, Felix
Frenkel, Hadar
Friedberger, Karlheinz
Frohme, Markus
Fu, Feisi
Fürnkranz, Johannes
Giacobbe, Mirco
Gjøl Jensen, Peter

Gossen, Frederik
Goudsmid, Ohad
Griggio, Alberto
Grover, Kush
Gutiérrez, Elena
Haaswijk, Winston
Hadžić, Vedad
Hahn, Ernst Moritz
Hansen, Mikkel
Hartmanns, Arnd
Hecking-Harbusch, Jesko
Hofmann, Jana
Holzner, Stephan
Huginin, Jasper
Humenberger, Andreas
Hupel, Lars
Hyvärinen, Antti
Irfan, Ahmed
Jasper, Marc
Jaulin, Luc
Jensen, Mathias Claus
Jensen, Peter Gjøl
Jonas, Martin
Jonsson, Bengt
Jonáš, Martin
Kacianka, Severin
Kaminski, Benjamin Lucien
Kanav, Sudeep
Kempa, Brian
Khalimov, Ayrat
Kiourti, Panagiota
Klauck, Michaela
Klüppelholz, Sascha
Koenighofer, Bettina
Kopetzki, Dawid
Krcal, Pavel
Kröger, Paul
Kupferman, Orna
Köhl, Maximilian
Lahkim Bennani, Ismail
Legay, Axel
Lemberger, Thomas
Liang, Chencheng

Lorber, Florian
Ma, Meiyi
Major, Juraj
Mann, Makai
Marcovich, Ron
Marescotti, Matteo
Martins, Ruben
Meggendorfer, Tobias
Mikučionis, Marius
Mitsch, Stefan
Mover, Sergio
Mues, Malte
Murtovi, Alnis
Möhlmann, Eike
Mömke, Tobias
Müller, David
Narváez, David
Naujokat, Stefan
Oliveira da Costa, Ana
Otoni, Rodrigo
Pagel, Jens
Parlato, Gennaro
Paskevich, Andrei
Peppelman, Marijn
Perelli, Giuseppe
Pivoluska, Matej
Popescu, Andrei
Puch, Stefan
Putot, Sylvie
Rebola-Pardo, Adrián

Reynolds, Andrew
Rothenberg, Bat-Chen
Roveri, Marco
Rowe, Reuben
Rüthing, Oliver
Schilling, Christian
Shoukry, Yasser
Spießl, Martin
Srba, Jiri
Stankovic, Miroslav
Stierand, Ingo
Štill, Vladimír
Stjerna, Albin
Stock, Gregory
Stojic, Ivan
Theel, Oliver
Tian, Chun
Tonetta, Stefano
Trtík, Marek
van der Ploeg, Atze
Vom Dorff, Sebastian
Wardega, Kacper
Weininger, Maximilian
Wendler, Philipp
Wimmer, Simon
Winkels, Jan
Yolcu, Emre
Zeljić, Aleksandar
Zhou, Weichao

Contents – Part II

Bisimulation

An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems	3
<i>David N. Jansen, Jan Friso Groote, Jeroen J. A. Keiren, and Anton Wijs</i>	
Verifying Quantum Communication Protocols with Ground Bisimulation. . . .	21
<i>Xudong Qin, Yuxin Deng, and Wenjie Du</i>	
Deciding the Bisimilarity of Context-Free Session Types	39
<i>Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos</i>	
Sharp Congruences Adequate with Temporal Logics Combining Weak and Strong Modalities	57
<i>Frédéric Lang, Radu Mateescu, and Franco Mazzanti</i>	

Verification and Efficiency

How Many Bits Does it Take to Quantize Your Neural Network?.	79
<i>Mirco Giacobbe, Thomas A. Henzinger, and Mathias Lechner</i>	
Highly Automated Formal Proofs over Memory Usage of Assembly Code. . . .	98
<i>Freek Verbeek, Joshua A. Bockenek, and Binoy Ravindran</i>	
GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts.	118
<i>Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio</i>	
CPU Energy Meter: A Tool for Energy-Aware Algorithms Engineering.	126
<i>Dirk Beyer and Philipp Wendler</i>	

Logic and Proof

Practical Machine-Checked Formalization of Change Impact Analysis	137
<i>Karl Palmskog, Ahmet Celik, and Milos Gligoric</i>	
What’s Decidable About Program Verification Modulo Axioms?	158
<i>Umang Mathur, P. Madhusudan, and Mahesh Viswanathan</i>	
Formalized Proofs of the Infinity and Normal Form Predicates in the First-Order Theory of Rewriting.	178
<i>Alexander Lochmann and Aart Middeldorp</i>	

Fold/Unfold Transformations for Fixpoint Logic	195
<i>Naoki Kobayashi, Grigory Fedyukovich, and Aarti Gupta</i>	
Tools and Case Studies	
Verifying OpenJDK’s LinkedList using KeY	217
<i>Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Marko van Eekelen, and Stijn de Gouw</i>	
Analysing installation scenarios of Debian packages	235
<i>Benedikt Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen</i>	
Endicheck: Dynamic Analysis for Detecting Endianness Bugs	254
<i>Roman Kápl and Pavel Parizek</i>	
Describing and Simulating Concurrent Quantum Systems	271
<i>Richard Bornat, Jaap Boender, Florian Kammüller, Guillaume Poly, and Rajagopal Nagarajan</i>	
EMTST: Engineering the Meta-theory of Session Types	278
<i>David Castro, Francisco Ferreira, and Nobuko Yoshida</i>	
Games and Automata	
Solving Mean-Payoff Games via Quasi Dominions	289
<i>Massimo Benerecetti, Daniele Dell’Erba, and Fabio Mogavero</i>	
Partial-Order Reduction for Parity Games with an Application on Parameterised Boolean Equation Systems	307
<i>Thomas Neele, Tim A. C. Willemse, and Wieger Wesselink</i>	
Polynomial Identification of ω-Automata	325
<i>Dana Angluin, Dana Fisman, and Yaara Shoval</i>	
SV-COMP 2020	
Advances in Automatic Software Verification: SV-COMP 2020	347
<i>Dirk Beyer</i>	
2LS: Heap Analysis and Memory Safety (Competition Contribution)	368
<i>Viktor Malik, Peter Schrammel, and Tomáš Vojnar</i>	
COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution)	373
<i>Willem Visser and Jaco Geldenhuys</i>	

DARTAGNAN: Bounded Model Checking for Weak Memory Models (Competition Contribution)	378
<i>Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer</i>	
VeriAbs : Verification by Abstraction and Test Generation (Competition Contribution)	383
<i>Mohammad Afzal, Supratik Chakraborty, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Ashutosh Gupta, Shrawan Kumar, Charles Babu M, Divyesh Unadkat, and R Venkatesh</i>	
GACAL: Conjecture-Based Verification (Competition Contribution)	388
<i>Benjamin Quiring and Panagiotis Manolios</i>	
Java Ranger at SV-COMP 2020 (Competition Contribution)	393
<i>Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser</i>	
JDART: Dynamic Symbolic Execution for JAVA Bytecode (Competition Contribution)	398
<i>Malte Mues and Falk Howar</i>	
Map2Check: Using Symbolic Execution and Fuzzing (Competition Contribution)	403
<i>Herbert Rocha, Rafael Menezes, Lucas C. Cordeiro, and Raimundo Barreto</i>	
PredatorHP Revamped (Not Only) for Interval-Sized Memory Regions and Memory Reallocation (Competition Contribution)	408
<i>Petr Peringer, Veronika Šoková, and Tomáš Vojnar</i>	
Symbiotic 7: Integration of Predator and More (Competition Contribution)	413
<i>Marek Chalupa, Tomáš Jašek, Lukáš Tomovič, Martin Hruška, Veronika Šoková, Paulína Ayaziová, Jan Strejček, and Tomáš Vojnar</i>	
Ultimate Taipan with Symbolic Interpretation and Fluid Abstractions (Competition Contribution)	418
<i>Daniel Dietsch, Matthias Heizmann, Alexander Nutz, Claus Schätzle, and Frank Schüssele</i>	
Author Index	423

Contents – Part I

Program Verification

Software Verification with PDR: An Implementation of the State of the Art.	3
<i>Dirk Beyer and Matthias Dangl</i>	
Verifying Array Manipulating Programs with Full-Program Induction	22
<i>Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat</i>	
Interpretation-Based Violation Witness Validation for C: NITWIT	40
<i>Jan Švejda, Philipp Berger, and Joost-Pieter Katoen</i>	
A Calculus for Modular Loop Acceleration	58
<i>Florian Frohn</i>	

SAT and SMT

Mind the Gap: Bit-vector Interpolation recast over Linear Integer Arithmetic	79
<i>Takamasa Okudono and Andy King</i>	
Automated and Sound Synthesis of Lyapunov Functions with SMT Solvers	97
<i>Daniele Ahmed, Andrea Peruffo, and Alessandro Abate</i>	
A Study of Symmetry Breaking Predicates and Model Counting.	115
<i>Wenxi Wang, Muhammad Usman, Alyas Almaawi, Kaiyuan Wang, Kuldeep S. Meel, and Sarfraz Khurshid</i>	
MUST: Minimal Unsatisfiable Subsets Enumeration Tool	135
<i>Jaroslav Bendik and Ivana Černá</i>	

Timed and Dynamical Systems

Safe Decomposition of Startup Requirements: Verification and Synthesis. . . .	155
<i>Alessandro Cimatti, Luca Geatti, Alberto Griggio, Greg Kimberly, and Stefano Tonetta</i>	
Multi-agent Safety Verification Using Symmetry Transformations.	173
<i>Hussein Sibai, Navid Mokhlesi, Chuchu Fan, and Sayan Mitra</i>	

Relational Differential Dynamic Logic	191
<i>Juraj Kolčák, Jérémy Dubut, Ichiro Hasuo, Shin-ya Katsumata, David Sprunger, and Akihisa Yamada</i>	
Verifying Concurrent Systems	
Assume, Guarantee or Repair	211
<i>Hadar Frenkel, Orna Grumberg, Corina Pasareanu, and Sarai Sheinvald</i>	
Structural Invariants for the Verification of Systems with Parameterized Architectures	228
<i>Marius Bozga, Javier Esparza, Radu Iosif, Joseph Sifakis, and Christoph Welzel</i>	
Automated Verification of Parallel Nested DFS	247
<i>Wytse Oortwijn, Marieke Huisman, Sebastiaan J. C. Joosten, and Jaco van de Pol</i>	
Discourje: Runtime Verification of Communication Protocols in Clojure	266
<i>Ruben Hamers and Sung-Shik Jongmans</i>	
Probabilistic Systems	
Scenario-Based Verification of Uncertain MDPs	287
<i>Murat Cubuktepe, Nils Jansen, Sebastian Junges, Joost-Pieter Katoen, and Ufuk Topcu</i>	
Good-for-MDPs Automata for Probabilistic Analysis and Reinforcement Learning	306
<i>Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak</i>	
Farkas Certificates and Minimal Witnesses for Probabilistic Reachability Constraints	324
<i>Florian Funke, Simon Jantsch, and Christel Baier</i>	
Simple Strategies in Multi-Objective MDPs	346
<i>Florent Delgrange, Joost-Pieter Katoen, Tim Quatmann, and Mickael Randour</i>	
Model Checking and Reachability	
Partial Order Reduction for Deep Bug Finding in Synchronous Hardware . . .	367
<i>Makai Mann and Clark Barrett</i>	

Revisiting Underapproximate Reachability for Multipushdown Systems	387
<i>S. Akshay, Paul Gastin, S Krishna, and Sparsa Roychowdhury</i>	
KReach: A Tool for Reachability in Petri Nets	405
<i>Alex Dixon and Ranko Lazić</i>	
AVR: Abstractly Verifying Reachability	413
<i>Aman Goel and Karem Sakallah</i>	
Timed and Probabilistic Systems	
Verified Certification of Reachability Checking for Timed Automata.	425
<i>Simon Wimmer and Joshua von Mutius</i>	
Learning One-Clock Timed Automata	444
<i>Jie An, Mingshuai Chen, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang</i>	
Rare Event Simulation for Non-Markovian Repairable Fault Trees	463
<i>Carlos E. Budde, Marco Biagi, Raúl E. Monti, Pedro R. D’Argenio, and Mariëlle Stoelinga</i>	
FIG: The Finite Improbability Generator	483
<i>Carlos E. Budde</i>	
MORA - Automatic Generation of Moment-Based Invariants	492
<i>Ezio Bartocci, Laura Kovács, and Miroslav Stankovič</i>	
Author Index	499

Bisimulation



An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems

David N. Jansen¹ , Jan Friso Groote² ,
Jeroen J.A. Keiren² , and Anton Wijs² 



¹ State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China dnjansen@ios.ac.cn

² Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands
{J.F.Groote, J.J.A.Keiren, A.J.Wijs}@tue.nl

Abstract. Branching bisimilarity is a behavioural equivalence relation on labelled transition systems (LTSs) that takes internal actions into account. It has the traditional advantage that algorithms for branching bisimilarity are more efficient than ones for other weak behavioural equivalences, especially weak bisimilarity. With m the number of transitions and n the number of states, the classic $O(mn)$ algorithm was recently replaced by an $O(m(\log |Act| + \log n))$ algorithm [9], which is unfortunately rather complex. This paper combines its ideas with the ideas from Valmari [20], resulting in a simpler $O(m \log n)$ algorithm. Benchmarks show that in practice this algorithm is also faster and often far more memory efficient than its predecessors, making it the best option for branching bisimulation minimisation and preprocessing for calculating other weak equivalences on LTSs.

Keywords: Branching bisimilarity · Algorithm · Labelled transition systems

1 Introduction

Branching bisimilarity [8] is an alternative to weak bisimilarity [17]. Both equivalences allow the reduction of labelled transition systems (LTSs) containing transitions labelled with internal actions, also known as silent, hidden or τ -actions.

One of the distinct advantages of branching bisimilarity is that, from the outset, an efficient algorithm has been available [10], which can be used to calculate whether two states are equivalent and to calculate a quotient LTS. It has complexity $O(mn)$ with m the number of transitions and n the number of states. It is more efficient than classic algorithms for weak bisimilarity, which use transitive closure (for instance, [16] runs in $O(n^2 m \log n + mn^{2.376})$, where $n^{2.376}$ is the time for computing the transitive closure), and algorithms for weak simulation equivalence (strong simulation equivalence can be computed in $O(mn)$ [12], and for weak simulation equivalence first the transitive closure needs to be computed). The algorithm is also far more efficient than algorithms for trace-based

equivalence notions, such as (weak) trace equivalence or weak failure equivalence [16].

Branching bisimilarity also enjoys the nice mathematical property that there exists a canonical quotient with a minimal number of states and transitions (contrary to, for instance, trace-based equivalences). Additionally, as branching bisimilarity is coarser than virtually any other behavioural equivalence taking internal actions into account [7], it is ideal for preprocessing. In order to calculate a desired equivalence, one can first reduce the behaviour modulo branching bisimilarity, before applying a dedicated algorithm on the often substantially reduced transition system. In the mCRL2 toolset [5] this is common practice.

In [9,11] an algorithm to calculate stuttering equivalence on Kripke structures with complexity $O(m \log n)$ was proposed. Stuttering equivalence essentially differs from branching bisimilarity in the fact that transitions do not have labels and as such all transitions can be viewed as internal. In these papers it was shown that branching bisimilarity can be calculated by translating LTSs to Kripke structures, encoding the labels of transitions into labelled states following [6,19]. This led to an $O(m(\log |Act| + \log n))$ or $O(m \log m)$ algorithm for branching bisimilarity.

Besides the time complexity, the algorithm in [9,11] has two disadvantages. First, the translation to Kripke structures introduces a new state and a new transition per action label and target state of a transition, which increases the memory required to calculate branching bisimilarity. This made it far less memory efficient than the classical algorithm of [10], and this was perceived as a substantial practical hindrance. For instance, when reducing systems consisting of tens of millions of states, such as [2], memory consumption is the bottleneck. Second, the algorithm in [9,11] is very complex. To illustrate the complexity, implementing it took approximately half a person-year.

Contributions. We present an algorithm for branching bisimilarity that runs directly on LTSs in $O(m \log n)$ time and that is simpler than the algorithm of [9,11]. To achieve this we use an idea from Valmari and Lehtinen [20,21] for strong bisimilarity. The standard Paige–Tarjan algorithm [18], which has $O(m \log n)$ time complexity for strong bisimilarity on Kripke structures, registers work done in a separate partition of states. Valmari [20] observed that this leads to complexity $O(m \log m)$ on LTSs and proposed to use a partition of transitions, whose elements he (and we) calls *bunches*, to register work done. This reduces the time complexity on LTSs to $O(m \log n)$.

Using this idea we design our more straightforward algorithm for branching bisimilarity on LTSs. Essentially, this makes the maintenance of action labels particularly straightforward and allows to simplify the handling of new, so-called, bottom states [10]. It also leads to a novel main invariant, which we formulate as Invariant 1. It allows us to prove the correctness of the algorithm in a far more straightforward way than before.

We have proven the correctness and complexity of the algorithm in detail [14] and demonstrate that it outperforms all preceding algorithms both in time and

space when the LTSs are sizeable. This is illustrated with more than 30 example LTSs. This shows that the new algorithm pushes the state-of-the-art in comparing and minimising the behaviour of LTSs w.r.t. weak equivalences, either directly (branching bisimilarity) or using the form of a preprocessing step (for other weak equivalences).

Despite the fact that this new algorithm is more straightforward than the previous $O(m(\log |Act| + \log n))$ algorithm [9], the implementation of the algorithm is still not easy. To guard against implementation errors, we extensively applied random testing, comparing the output with that of other algorithms. The algorithms and their source code are freely available in the mCRL2 toolset [5].

Overview of the article. In Section 2 we provide the definition of LTSs and branching bisimilarity. In Section 3 we provide the core algorithm with high-level data structures, correctness and complexity. The subsequent section presents the procedure for splitting blocks, which can be presented as an independent pair of coroutines. Section 5 presents some benchmarks. Proofs and implementation details are omitted in this paper, and can be found in [14].

2 Branching bisimilarity

In this section we define labelled transition systems and branching bisimilarity.

Definition 1 (Labelled transition system). A labelled transition system (LTS) is a triple $A = (S, Act, \rightarrow)$ where

1. S is a finite set of states. The number of states is denoted by n .
2. Act is a finite set of actions including the internal action τ .
3. $\rightarrow \subseteq S \times Act \times S$ is a transition relation. The number of transitions is necessarily finite and denoted by m .

It is common to write $t \xrightarrow{a} t'$ for $(t, a, t') \in \rightarrow$. With slight abuse of notation we write $t \xrightarrow{a} t' \in T$ instead of $(t, a, t') \in T$ for $T \subseteq \rightarrow$. We also write $t \xrightarrow{a} Z$ for the set of transitions $\{t \xrightarrow{a} t' \mid t' \in Z\}$, and $Z \xrightarrow{a} Z'$ for the set $\{t \xrightarrow{a} t' \mid t \in Z, t' \in Z'\}$. We call all actions except τ the *visible* actions. If $t \xrightarrow{a} t'$, we say that from t , the state t' , the action a , and the transition $t \xrightarrow{a} t'$ are *reachable*.

Definition 2 (Branching bisimilarity). Let $A = (S, Act, \rightarrow)$ be an LTS. We call a relation $R \subseteq S \times S$ a branching bisimulation relation iff it is symmetric and for all $s, t \in S$ such that $s R t$ and all transitions $s \xrightarrow{a} s'$ we have:

1. $a = \tau$ and $s' R t$, or
2. there is a sequence $t \xrightarrow{\tau} \dots \xrightarrow{\tau} t' \xrightarrow{a} t''$ such that $s R t'$ and $s' R t''$.

Two states s and t are branching bisimilar, denoted by $s \xleftrightarrow{b} t$, iff there is a branching bisimulation relation R such that $s R t$.

Note that branching bisimilarity is an equivalence relation. Given an equivalence relation R , a transition $s \xrightarrow{a} t$ is called *inert* iff $a = \tau$ and $s R t$. If $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} t_{n-1} \xrightarrow{\tau} t_n \xrightarrow{a} t'$ such that $t R t_i$ for $1 \leq i \leq n$, we say that the state t_n , the action a , and the transition $t_n \xrightarrow{a} t'$ are *inertly reachable* from t .

The equivalence classes of branching bisimilarity partition the set of states.

Definition 3 (Partition). For a set X a partition Π of X is a disjoint cover of X , i.e., $\Pi = \{B_i \subseteq X \mid B_i \neq \emptyset, 1 \leq i \leq k\}$ such that $B_i \cap B_j = \emptyset$ for all $1 \leq i < j \leq k$ and $X = \bigcup_{1 \leq i \leq k} B_i$.

A partition Π' is a refinement of Π iff for every $B' \in \Pi'$ there is some $B \in \Pi$ such that $B' \subseteq B$.

We will often use that a partition Π induces an equivalence relation in the following way: $s \equiv_{\Pi} t$ iff there is some $B \in \Pi$ containing both s and t .

3 The algorithm

In this section we present the core algorithm. In the next section we deal with the actual splitting of blocks in the partition. We start off with an abstract description of this core part.

3.1 High-level description of the algorithm

The algorithm is a partition refinement algorithm. It iteratively refines two partitions Π_s and Π_t . Partition Π_s is a partition of states in S that is coarser than branching bisimilarity. We refer to the elements of Π_s as *blocks*, typically denoted using B . Partition Π_t partitions the non-inert transitions of \rightarrow , where inertness is interpreted with respect to \equiv_{Π_s} . We refer to the elements of Π_t as *bunches*, typically denoted using T .

The partition of transitions Π_t records the current knowledge about transitions. Transitions are in different bunches iff the algorithm has established that they cannot simulate each other (i.e., they cannot serve as $s \xrightarrow{a} s'$ and $t' \xrightarrow{a} t''$ in Definition 2).

The partition of states Π_s records the current knowledge about branching bisimilarity. Two states are in different blocks iff the algorithm has found a proof that they are not branching bisimilar (this is formalised in Invariant 3). This implies that Π_s must be such that states with outgoing transitions in different combinations of bunches are in different blocks (Invariant 1).

Before performing partition refinement, the LTS is preprocessed to contract τ -strongly connected components (SCCs) into a single state without a τ -loop. This step is valid as all states in a τ -SCC are branching bisimilar. Consequently, every block has *bottom states*, i.e., states without outgoing inert τ -transitions [10].

The core invariant of the algorithm says that if one state in a block can inertly reach a transition in a bunch, all states in that block can inertly reach a transition in this bunch. This can be formulated in terms of bottom states:

Invariant 1 (Bunches). Π_s is stable under Π_t , i.e., if a bunch $T \in \Pi_t$ contains a transition with its source state in a block $B \in \Pi_s$, then every bottom state in block B has a transition in bunch T .

The initial partitions Π_s and Π_t are the coarsest partitions that satisfy Invariant 1. Π_t starts with a single bunch consisting of all non-inert transitions. Then, in Π_s we need to separate states with some transition in this bunch from those without. We define B_{vis} to be the set of states from which a visible transition is inertly reachable, and B_{invis} to be the other states. Then $\Pi_s = \{B_{\text{vis}}, B_{\text{invis}}\} \setminus \{\emptyset\}$.

Transitions in a bunch may have different labels or go to different blocks. In that case, the bunch can be split as these transitions cannot simulate each other. If we manage to achieve the situation where all transitions in a bunch have the same label and go to the same target block, the obtained partition turns out to be a branching bisimulation. Therefore, we want to split each bunch into so-called action-block-slices defined below. We also immediately define some other sets derived from Π_t and Π_s as we require them in our further exposition. So, we have:

- The *action-block-slices*, i.e., the transitions in T with label a ending in B' :

$$T_{\xrightarrow{a} B'} = \{s \xrightarrow{a} s' \in T \mid s' \in B'\}.$$
- The *block-bunch-slices*, i.e., the transitions in T starting in B :

$$T_{B \rightarrow} = \{s \xrightarrow{b} s' \in T \mid s \in B\}.$$
- A block-bunch-slice intersected with an action-block-slice:

$$T_{B \xrightarrow{a} B'} = T_{B \rightarrow} \cap T_{\xrightarrow{a} B'} = \{s \xrightarrow{a} s' \in T \mid s \in B \wedge s' \in B'\}.$$
- The *bottom states* of B , i.e., the states without outgoing inert transitions:

$$\text{Bottom}(B) = \{s \in B \mid \neg \exists s' \in B. s \xrightarrow{\tau} s'\}.$$
- The states in B with a transition in bunch T : $B \xrightarrow{\tau} = \{s \mid s \xrightarrow{a} s' \in T_{B \rightarrow}\}.$
- The outgoing transitions of block B : $B_{\rightarrow} = \{s \xrightarrow{a} s' \mid s \in B, a \in \text{Act}, s' \in S\}.$
- The incoming transitions of block B : $B_{\leftarrow} = \{s \xrightarrow{a} s' \mid s \in S, a \in \text{Act}, s' \in B\}.$

The block-bunch-slices and action-block-slices are explicitly maintained as auxiliary data structures in the algorithm in order to meet the required performance bounds. If the partitions Π_s or Π_t are adapted, all the derived sets above also change accordingly.

A bunch can be *trivial*, which means that it only contains one action-block-slice, or it can contain multiple action-block-slices. In the latter case one action-block-slice is split off to become a bunch by itself. However, this may invalidate Invariant 1. Some states in a block may only have transitions in the new bunch while other states have only transitions in the old bunch. Therefore, blocks have to be split to satisfy Invariant 1. Splitting blocks can cause bunches to become non-trivial because action-block-slices fall apart.

This splitting is repeated until all bunches are trivial, and as already stated above, the obtained partition Π_s is the required branching bisimulation. As the transition system is finite this process of repeated splitting terminates.

3.2 Abstract algorithm

We first present an abstract version of the algorithm in Algorithm 1. Its behaviour is as follows. As long as there are non-trivial bunches—i.e, bunches containing multiple action-block-slices—, these bunches need to be split such that they ultimately become trivial. The outer loop (Lines 1.2–1.19) takes a

Algorithm 1 Abstract algorithm for branching bisimulation partitioning

```

1.1: Contract  $\tau$ -SCCs; initialize  $\Pi_s$  and  $\Pi_t$ 
1.2: for all non-trivial bunches  $T \in \Pi_t$  do
1.3:   Select an action-block-slice  $T_{\alpha, B'} \subset T$ 
1.4:   Split  $T$  into  $T_{\alpha, B'}$  and  $T \setminus T_{\alpha, B'}$ 
1.5:   for all unstable blocks  $B \in \Pi_s$  (i.e.,  $\emptyset \neq T_{B \xrightarrow{\alpha} B'} \neq T_{B \rightarrow}$ ) do
1.6:     First make  $T_{B \xrightarrow{\alpha} B'}$  a primary splitter; then make  $T_{B \rightarrow} \setminus T_{B \xrightarrow{\alpha} B'}$  a secondary splitter
1.7:   end for
1.8:   for all splitters  $T'_{B \rightarrow}$  (in order) do
1.9:     Split  $B$  into the subblock  $R$  that can inertly reach  $T'_{B \rightarrow}$  and the rest  $U$ 
1.10:    if  $T'_{B \rightarrow}$  was a primary splitter (note:  $T'_{B \rightarrow} = T_{B \xrightarrow{\alpha} B'}$ ) then
1.11:      Make  $T_{U \rightarrow} \setminus T_{U \xrightarrow{\alpha} B'}$  a non-splitter
1.12:    end if
1.13:    if there are new non-inert transitions  $R \xrightarrow{\tau} U$  then
1.14:      Split  $R$  into the subblock  $N$  that can inertly reach  $R \xrightarrow{\tau} U$  and the rest  $R'$ 
1.15:      Make all block-bunch-slices  $T_{N \rightarrow}$  of  $N$  secondary splitters
1.16:      Create a bunch for the new non-inert transitions  $(N \xrightarrow{\tau} U) \cup (N \xrightarrow{\tau} R')$ 
1.17:    end if
1.18:  end for
1.19: end for
1.20: return  $\Pi_s$ 

```

non-trivial bunch T from Π_t , and from this it moves an action-block-slice $T_{\alpha, B'}$ into its own bunch in Π_t (Line 1.4). Hence, bunch T is reduced to $T \setminus T_{\alpha, B'}$.

The two new bunches $T_{\alpha, B'}$ and $T \setminus T_{\alpha, B'}$ can cause instability, violating Invariant 1. This means there can be blocks with transitions in one new bunch, but some bottom states only have transitions in the other new bunch. For such blocks, stability needs to be restored by splitting them.

To restore this stability we investigate all block-bunch-slices in one of the new bunches, namely $T_{\alpha, B'}$. Blocks that do not have transitions in these block-bunch-slices are stable with respect to both bunches. To keep track of the blocks that still need to be split, we partition the block-bunch-slices $T_{B \rightarrow}$ into *stable* and *unstable* block-bunch-slices. A block-bunch-slice is stable if we have ensured that it is not a splitter for any block. Otherwise it is deemed unstable, and it needs to be checked whether it is stable, or whether the block B must be split. The first inner loop (Lines 1.5–1.7) inserts all unstable block-bunch-slices into the *splitter list*. Block-bunch-slices of the shape $T_{B \xrightarrow{\alpha} B'}$ in the splitter list are labelled *primary*, and other list entries are labelled *secondary*.

In the second loop (Lines 1.8–1.18), one splitter $T'_{B \rightarrow}$ from the splitter list is taken at a time and its source block is split into R (the part that can inertly reach $T'_{B \rightarrow}$) and U (the part that cannot inertly reach $T'_{B \rightarrow}$) to re-establish stability.

If $T'_{B \rightarrow}$ was a primary splitter of the form $T_{B \xrightarrow{\alpha} B'}$, then we know that U must be stable under $T_{U \rightarrow} \setminus T_{U \xrightarrow{\alpha} B'}$, as every bottom state in B has a transition in the former block-bunch-slice $T_{B \rightarrow}$, and as the states in U have no transition in $T_{B \xrightarrow{\alpha} B'}$, every bottom state in U must have a transition in $T_{B \rightarrow} \setminus T_{B \xrightarrow{\alpha} B'}$. Therefore, at Line 1.11, block-bunch-slice $T_{U \rightarrow} \setminus T_{U \xrightarrow{\alpha} B'}$ can be removed from the splitter list. This is the three-way split from [18].

Some inert transitions may have become non-inert, namely the τ -transitions that go from R to U . There cannot be τ -transitions from U to R . The new non-inert transitions were not yet part of a bunch in Π_t . So, a new bunch $R \xrightarrow{\tau} U$ is formed for them. All transitions in this new bunch leave R and thus R is the only

block that may not be stable under this new bunch. To avoid superfluous work, we split off the unstable part N , i.e. the part that can inertly reach a transition in $R \xrightarrow{\tau} U$ and contains all new bottom states, at Line 1.14. The original bottom states of R become the bottom states of R' . There can be transitions $N \xrightarrow{\tau} R'$ that also become non-inert, and we add these to the new bunch $R \xrightarrow{\tau} U$. As observed in [10], blocks containing new bottom states can become unstable under any bunch. So, stability of N (but not of R') must be re-established, and all block-bunch-slices leaving N are put on the splitter list at Line 1.15.

3.3 Correctness

The validity of the algorithm follows from a number of major invariants. The main invariant, Invariant 1, is valid at Line 1.2. Additionally, the algorithm satisfies the following three invariants.

Invariant 2 (Bunches are not unnecessarily split). *For any pair of non-inert transitions $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$, if $s, t \in B$ and $s', t' \in B'$ then $s \xrightarrow{a} s' \in T$ and $t \xrightarrow{a} t' \in T$ for some bunch $T \in \Pi_t$.*

Invariant 3 (Preservation of branching bisimilarity). *For all states $s, t \in S$, if $s \xleftrightarrow{b} t$, then there is some block $B \in \Pi_s$ such that $s, t \in B$.*

Invariant 4 (No inert loops). *There is no inert loop in a block, i.e., for every sequence $s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \dots \xrightarrow{\tau} s_n$ with $s_i \in B \in \Pi_s$, $n > 1$ it holds that $s_1 \neq s_n$.*

Invariant 2 indicates that two non-inert transitions that (1) start in the same block, (2) have the same label, and (3) end in the same block, always reside in the same bunch. Invariant 3 says that branching bisimilar states never end up in separate blocks. Invariant 4 ensures that all τ -paths in each block are finite. As a consequence every block has at least one bottom state, and from every state a bottom state can be inertly reached.

The invariants given above allow us to prove that the algorithm works correctly. When the algorithm terminates (and this always happens, see Section 3.5), branching bisimilar states are perfectly grouped in blocks.

Theorem 1. *From the Invariants 1, 3 and 4, it follows that after the algorithm terminates, $\equiv_{\Pi_s} = \xleftrightarrow{b}$.*

Because of the space restrictions here, the proofs are omitted. The interested reader is referred to [14] for the details.

3.4 In-depth description of the algorithm

To show that the algorithm has the desired $O(m \log n)$ time complexity, we now give a more detailed description of the algorithm. The pseudocode of the detailed algorithm is given in Algorithm 2. This algorithm serves two purposes. First of all, it clarifies how the data structures are used, and refines many of the steps in the high-level algorithm. Additionally, time budgets for parts of the algorithm

Algorithm 2 Detailed algorithm for branching bisimulation partitioning

```

2.1: Find  $\tau$ -SCCs and contract each of them to a single state
2.2:  $B_{\text{vis}} := \{s \in S \mid s \text{ can inertly reach some } s' \xrightarrow{a} s'\}$ ;  $B_{\text{invis}} := S \setminus B_{\text{vis}}$ 
2.3:  $\Pi_s := \{B_{\text{vis}}, B_{\text{invis}}\} \setminus \{\emptyset\}$ 
2.4:  $\Pi_t := \{\{s \xrightarrow{a} s' \mid a \in \text{Act} \setminus \{\tau\}, s, s' \in S\} \cup B_{\text{vis}} \xrightarrow{\tau} B_{\text{invis}}\}$ 
2.5: for all non-trivial bunches  $T \in \Pi_t$  do
2.6:   Select  $a \in \text{Act}$  and  $B' \in \Pi_s$  with  $|T_{a \rightarrow B'}| \leq \frac{1}{2} |T|$ 
2.7:    $\Pi_t := (\Pi_t \setminus \{T\}) \cup \{T_{a \rightarrow B'}, T \setminus T_{a \rightarrow B'}\}$ 
2.8:   for all unstable blocks  $B \in \Pi_s$  with  $\emptyset \subset T_{B \xrightarrow{a} B'} \subset T_{B \rightarrow}$  do
2.9:     Append  $T_{B \xrightarrow{a} B'}$  as primary to the splitter list
2.10:    Append  $T_{B \rightarrow} \setminus T_{B \xrightarrow{a} B'}$  as secondary to the splitter list
2.11:    Mark all transitions in  $T_{B \xrightarrow{a} B'}$ 
2.12:    For every state  $\in B$  with both marked outgoing transitions
        and outgoing transitions in  $T_{B \rightarrow} \setminus T_{B \xrightarrow{a} B'}$ , mark one such
        transition
2.13:   end for
2.14:   for all splitters  $T'_{B \rightarrow}$  in the splitter list (in order) do
2.15:      $\langle R, U \rangle := \text{split}(B, T'_{B \rightarrow})$ 
2.16:     Remove  $T'_{B \rightarrow} = T_{R \rightarrow}$  from the splitter list
2.17:      $\Pi_s := (\Pi_s \setminus \{B\}) \cup (\{R, U\} \setminus \{\emptyset\})$ 
2.18:     if  $T_{B \rightarrow}$  was a primary splitter (note:  $T'_{B \rightarrow} = T_{B \xrightarrow{a} B'}$ ) then
2.19:       Remove  $T_{U \rightarrow} \setminus T_{U \xrightarrow{a} B'}$  from the splitter list
2.20:     end if
2.21:     if  $R \xrightarrow{\tau} U \neq \emptyset$  then
2.22:       Create a new bunch containing exactly  $R \xrightarrow{\tau} U$ , add
        $R \xrightarrow{\tau} U = (R \xrightarrow{\tau} U)_{R \rightarrow}$  to the splitter list, and mark
       all its transitions
2.23:        $\langle N, R' \rangle := \text{split}(R, R \xrightarrow{\tau} U)$ 
2.24:       Remove  $R \xrightarrow{\tau} U = (R \xrightarrow{\tau} U)_{N \rightarrow}$  from the splitter list
2.25:        $\Pi_s := (\Pi_s \setminus \{R\}) \cup (\{N, R'\} \setminus \{\emptyset\})$ 
2.26:       Add  $N \xrightarrow{\tau} R'$  to the bunch containing  $R \xrightarrow{\tau} U$ 
2.27:       Insert all  $T_{N \rightarrow}$  as secondary into the splitter list
2.28:       For each bottom state  $\in N$ , mark one of its outgoing
       transitions in every  $T_{N \rightarrow}$  where it has one
2.29:     end if
2.30:   end for
2.31: end for
2.32: return  $\Pi_s$ 

```

are printed in grey at the right-hand side of the pseudocode. We use these time budgets in Section 3.5 to analyse the overall complexity of the algorithm. We focus on the most important details in the algorithm.

At Lines 2.6–2.7, a *small* action-block-slice $T_{a \rightarrow B'}$ is moved into its own bunch, and T is reduced to $T \setminus T_{a \rightarrow B'}$. All blocks that have transitions in the two new bunches are added to the splitter list in Lines 2.8–2.13. This loop also marks some transitions (in the time complexity annotations we write $\text{Marked}(T_{B \rightarrow})$ for the marked transitions of block-bunch-slice $T_{B \rightarrow}$). The function of this marking is similar to that of the counters in [18]: it serves to determine quickly whether a bottom state has a transition in a secondary splitter $T_{B \rightarrow} \setminus T_{B \xrightarrow{a} B'}$ (or slices that are the result of splitting this slice). In general, a bottom state has transitions in some splitter block-bunch-slice if and only if it has marked transitions in this slice. There is one exception: After splitting under a primary splitter $T_{B \rightarrow}$, bottom states in U are not marked. But as they always have a transition in $T_{U \rightarrow} \setminus T_{U \xrightarrow{a} B'}$, U is already stable in this case (see Line 2.19).

The second loop is refined to Lines 2.14–2.30. In every iteration one splitter $T'_{B \rightarrow}$ from the splitter list is considered, and its source block is first split into R

and U . Formally, the routine $\text{split}(B, T)$ delivers the pair $\langle R, U \rangle$ defined by:

$$\begin{aligned} R &= \{s \in B \mid s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} s_n \xrightarrow{a} s' \text{ where } s_1, \dots, s_n \in B, s_n \xrightarrow{a} s' \in T\}, \\ U &= B \setminus R. \end{aligned} \tag{1}$$

We detail its algorithm and discuss its correctness in Section 4.

In Lines 2.21–2.28, the situation is handled when some inert transitions have become non-inert. We mark one of the outgoing transitions of every new bottom state such that we can find the bottom states with a transition in $T_{N \rightarrow}$ in time proportional to the number of such new bottom states.

We illustrate the algorithm in the following example. Note this also illustrates some of the details of the split subroutine, which is discussed in detail in Section 4.

Example 1. Consider the situation in Figure 1a. Observe that block B is stable w.r.t. the bunches T and T' . We have split off a small bunch $T_{\xrightarrow{a} B'}$ from T , and as a consequence, B needs to be restabilised. The bunches put on the splitter list initially are $T_{\xrightarrow{a} B'}$ and $T \setminus T_{\xrightarrow{a} B'}$. When putting these bunches on the splitter list, all transitions in $T_{B \xrightarrow{a} B'}$ are marked, see the m 's in Figure 1b. Also, for states that have transitions both in $T_{\xrightarrow{a} B'}$ and in $T \setminus T_{\xrightarrow{a} B'}$, one transition in the latter bunch is marked, see the m 's in Figure 1b.

We now first split B w.r.t. the primary splitter $T_{\xrightarrow{a} B'}$ into R , the states that can inertly reach $T_{\xrightarrow{a} B'}$, and U , the states that cannot. In Figure 1b, the states known to be destined for R are indicated by \oplus , the states known to be destined for U are indicated by \ominus . Initially, all states with a marked outgoing transition are destined for R , the remaining bottom state of B is destined for U . The split subroutine proceeds to extend sets R and U in a backwards fashion using two coroutines, marking a state destined for R if one of its successors is already in R , and marking a state destined for U if all its successors are in U . Here, the state in U does not have any incoming inert transitions, so its coroutine immediately terminates and all other states belong to R . Block B is split into subblocks R and U , as shown in Figure 1c. Block U is stable w.r.t. both $T_{\xrightarrow{a} B'}$ and $T \setminus T_{\xrightarrow{a} B'}$.

We still need to split R w.r.t. $T \setminus T_{\xrightarrow{a} B'}$, into R_1 and U_1 , say. For this, we use the marked transitions in $T \setminus T_{\xrightarrow{a} B'}$ as a starting point to compute all bottom states that can reach a transition in $T \setminus T_{\xrightarrow{a} B'}$. This guarantees that the time we use is proportional to the size of $T_{\xrightarrow{a} B'}$. Initially, there is one state destined for R_1 , marked \oplus in Figure 1c, and one state destined for U_1 , marked \ominus in the same figure. We now perform the two coroutines in split simultaneously. Figure 1d shows the situation after both coroutines have considered one transition: The U_1 -coroutine (which calculates the states that cannot inertly reach $T \setminus T_{\xrightarrow{a} B'}$) has initialised the counter *untested* of one state to 2 on Line 3.9ℓ of Algorithm 3 because two of its outgoing inert transitions have not yet been considered. The R_1 -coroutine (which calculates the states that can inertly reach $T \setminus T_{\xrightarrow{a} B'}$) has checked the unmarked transition in the splitter $T_{R \rightarrow} \setminus T_{R \xrightarrow{a} B'}$. As the latter coroutine has finished visiting unmarked transitions in the splitter, the U_1 -coroutine no longer needs to run the slow test loop at Lines 3.13ℓ–3.17ℓ of the left column of Algorithm 3. In Figure 1e the situation is shown after two more steps in the coroutines. Each has visited two extra transitions. There two

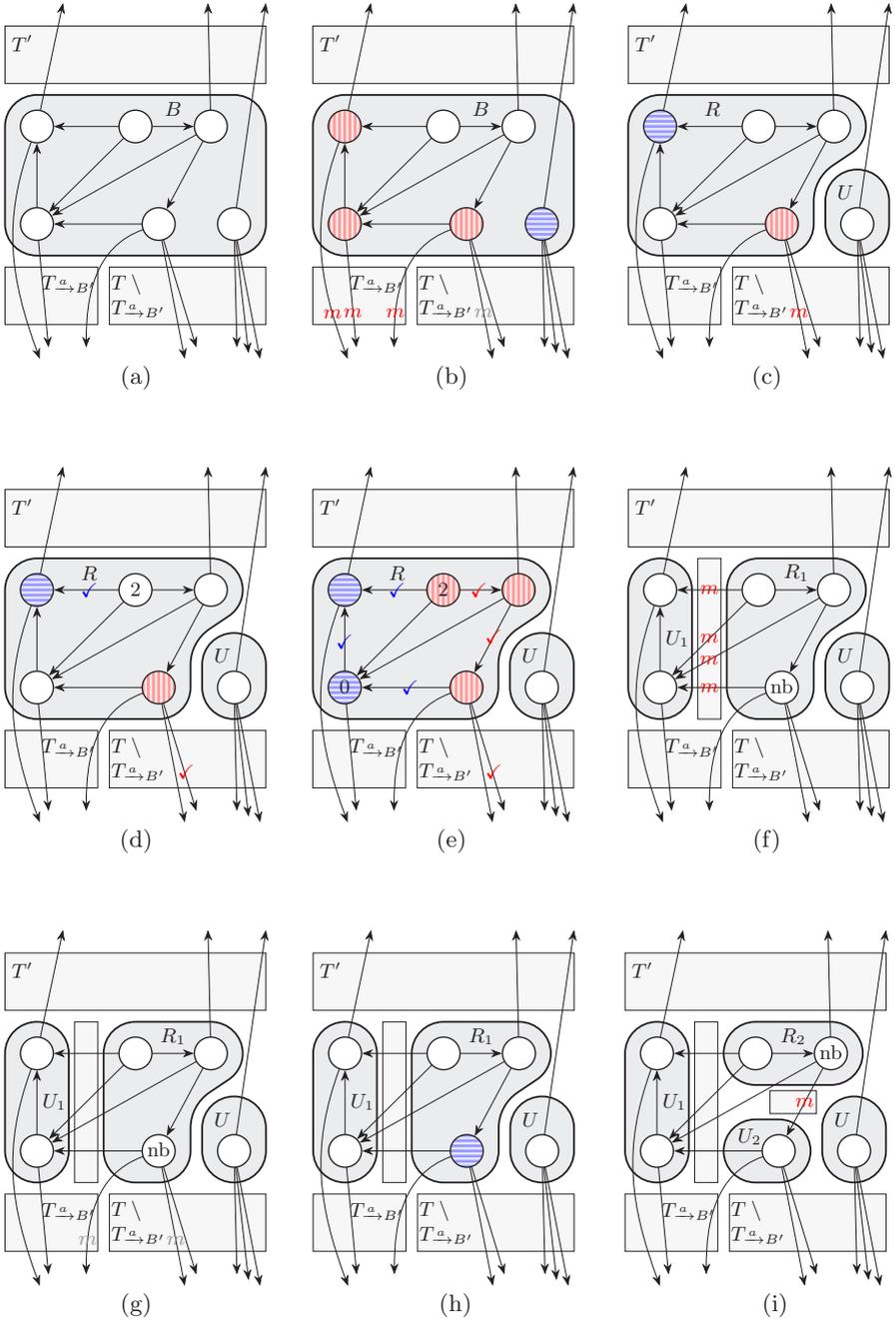


Fig. 1: Illustration of splitting of a small block from T and stabilising block B with respect to the new bunches $T^a_{>B'}$ and $T \setminus T^a_{>B'}$, as explained in Example 1.

extra are states destined for R_1 , marked \oplus , and one state is destined for U_1 with 0 remaining inert transitions, for which we know immediately that it has no transition in $T \setminus T_{\xrightarrow{\alpha} B'}$, this is marked \ominus . Now, the R_1 -coroutine is terminated, since it contains more than $\frac{1}{2}|R|$ states, and the remaining incoming transitions of states in U_1 are visited. This will not further extend U_1 . The result of splitting is shown in Figure 1f. Some inert transitions become non-inert, so a new bunch with transitions $R_1 \xrightarrow{\tau} U_1$ is created, and all these transitions are marked m .

We next have to split R_1 with respect to this new bunch into the set of states N_1 that can inertly reach a transition in the new bunch, and the set R'_1 that cannot inertly reach this bunch. In this case, all states in R_1 have a marked outgoing transition, hence $N_1 = R_1$, and $R'_1 = \emptyset$. The coroutine that calculates the set of states that cannot inertly reach a transition in the bunch will immediately terminate because there are no transitions to be considered.

Observe that $R_1 (= N_1)$ has a new bottom state, marked ‘nb’. This means that stability of R_1 with respect to any bunch is not guaranteed any more and needs to be re-established. We therefore consider all bunches in which R_1 has an outgoing transition. We add $T_{R_1 \xrightarrow{\alpha} B'}$, $T_{R_1 \rightarrow} \setminus T_{R_1 \xrightarrow{\alpha} B'}$ and $T'_{R_1 \rightarrow}$ to the splitter list as secondary splitters, and mark one outgoing transition from each bottom state in each of these bunches using m . This situation is shown in Figure 1g.

In this case, R_1 is stable w.r.t. $T_{R_1 \xrightarrow{\alpha} B'}$ and $T_{R_1 \rightarrow} \setminus T_{R_1 \xrightarrow{\alpha} B'}$, i.e., all states in R_1 can inertly reach a transition in both bunches. In both cases this is observed immediately after initialisation in split, since the set of states that cannot inertly reach a transition in these bunches is initially empty, and the corresponding coroutine terminates immediately.

Therefore, consider splitting R_1 with respect to $T'_{R_1 \rightarrow}$. This leads to R_2 , the set of states that can inertly reach a transition in T' , and U_2 , the set of states that cannot inertly reach a transition in T' . Note there are no marked transitions in $T'_{R_1 \rightarrow}$, so initially all bottom states of R_1 are destined for U_2 (marked \ominus in Figure 1h), and there are no states destined for R_2 . Then we start splitting R_1 . In the R_2 -coroutine, we first add the states with an unmarked transition in $T'_{R_1 \rightarrow}$ to R_2 at Line 3.4r (i.e., in the right column of Algorithm 3) and then all predecessors of the new bottom state need to be considered. When split terminates, there will be no additional states in U_2 , and the remaining states end up in R_2 .

The situation after splitting R_1 into R_2 and U_2 is shown in Figure 1i. One of the inert transitions (marked m) becomes non-inert. Furthermore, R_2 contains a new bottom state. This is the state with a transition in T' . As each block must have a bottom state, a non-bottom state had to become a bottom state.

We need to continue stabilising R_2 w.r.t. bunch $R_2 \xrightarrow{\tau} U_2$, which does not lead to a new split, and we need to restabilise R_2 w.r.t. all bunches in which it has an outgoing transition. This also does not lead to new splits, so the situation in Figure 1i after removing the markings is the final result of splitting.

3.5 Time complexity

Throughout this section, let n be the number of states and m the number of transitions in the LTS. To simplify the complexity notations we assume that

$n \leq m + 1$. This is not a significant restriction, since it is satisfied by any LTS in which every non-initial state has an incoming transition. We also write $in(s)$ and $out(s)$ for the sets of incoming and outgoing transitions of state s .

We use the principle ‘‘Process the smaller half’’ [13]: when a set is split into two parts, we spend time proportional to the size of the smaller subset. This leads to a logarithmic number of operations assigned to each element. We apply this principle twice, once to new bunches and once to new subblocks. Additionally, we spend some time on new bottom states. This is formulated in the following theorem.

Theorem 2. *For the main loop of Algorithm 2 we have:*

1. *A transition is moved to a new small bunch at most $\lceil \log_2 n^2 \rceil + 1$ times. Whenever this happens, constant time is spent on this transition.*
2. *A state s is moved to a new small subblock at most $\lceil \log_2 n \rceil$ times. Whenever this happens, $O(|in(s)| + |out(s)| + 1)$ time is spent on state s .*
3. *A state s becomes a new bottom state at most once. When this happens, $O(|out(s)| + 1)$ time is spent on state s .*

Summing up these time budgets leads to an overall time complexity of $O(m \log n)$.

These runtimes are annotated as time budgets in the main loop of Algorithm 2. Line 2.7 moves the transitions of $T_{a \rightarrow B'}$ to their new bunch, and Lines 2.6–2.14 take time proportional to the size of this new bunch.

A new subblock is formed at Line 2.17 (and at the same time, some states in subblock R may become new bottom states). Lines 2.15–2.22 take time proportional to its incoming and outgoing transitions. Similarly, a new subblock is formed in Line 2.23, and Lines 2.23–2.26 take time proportional to this subblock’s transitions.

Finally, new bottom states found in R (and separated into N) allow to spend time proportional to $Bottom(N)_{\rightarrow}$ at Lines 2.15–2.28. At Line 2.27 we need to include not only the current new bottom states but also the future ones because there may be block-bunch-slices that only have transitions from non-bottom states. When N is split under such a block-bunch-slice, at least one of these states will become a bottom state.

Time spent per marked transition fits the time bound because only a small number of transitions is marked: In Lines 2.11 and 2.12, at most two transitions are marked per transition in the small splitter $T_{a \rightarrow B'}$. Line 2.22 marks $R \xrightarrow{\tau} U \subseteq out(R) \cap in(U)$, which is always within the transitions of the smaller subblock. Line 2.28 marks no more transitions than the new bottom states have.

The initialisation in Lines 2.1–2.5 can be performed in $O(m)$ time, where the assumption $n \leq m + 1$ is used. Furthermore, we assume that we can access action labels fast enough to bucket sort the transitions in time $O(m)$, which is for instance the case if the action labels are consecutively numbered.

To meet the indicated time budgets, our implementation uses a number of data structures. States are stored in a *refinable partition* [21], grouped per block, in such a way that we can visit bottom states without spending time on non-bottom states. Transitions are stored in four linked refinable partitions, grouped

per source state, per target state, per bunch, and per block-bunch-slice, in such a way that we can visit marked transitions without spending time on unmarked transitions of the block. How these data structures are instrumental for the complexity can be found in [14].

4 Splitting blocks

The function $\text{split}(B, T)$, presented in Algorithm 3, refines block B into subblocks R and U , where R contains those states in B that can inertly reach a transition in T , and U contains the states that cannot, as formally specified in Equation (1).

These two sets are computed by two coroutines executing in lockstep: the two coroutines start the same number of loop iterations, so that the overhead is at most proportional to the faster of the two and all work done in both coroutines can be attributed to the smaller of the two subblocks R and U .

As a precondition, split requires that bottom states of B with an outgoing transition in $T_{B \rightarrow}$ have a marked outgoing transition in $T_{B \rightarrow}$. Formally, $\text{Bottom}(B) \xrightarrow{\text{Marked}(T_{B \rightarrow})} = \text{Bottom}(B) \xrightarrow{T_{B \rightarrow}}$. This allows to compute the initial sets: All states in $B \xrightarrow{\text{Marked}(T)}$, i.e., sources of marked transitions in T , are put in R . All bottom states that are not initially in R are put in U .

The sets are extended as follows in the coroutines. For R , first the states in $B \xrightarrow{T \setminus \text{Marked}(T)}$ are added that were not yet in R . These are all the sources of unmarked transitions in T . Using backward reachability along inert transitions, R is extended until no more states can be added.

Algorithm 3 Refinement of a block under a splitter

<pre> 3.1: function split(block B, block-bunch-slice T) 3.2: $R := B \xrightarrow{\text{Marked}(T)}$; $U := \text{Bottom}(B) \setminus R$ 3.3: begin coroutines 3.4: Set $\text{untested}[t]$ to undefined for all $t \in B$ 3.5: for all $s \in U$ while $U \leq \frac{1}{2} B$ do 3.6: for all inert $t \xrightarrow{\tau} s$ do 3.7: if $t \in R$ then Skip t, i.e. goto 3.6ℓ 3.8: if $\text{untested}[t]$ is undefined then 3.9: $\text{untested}[t] := \{t \xrightarrow{\tau} u \mid u \in B\}$ 3.10: end if 3.11: $\text{untested}[t] := \text{untested}[t] - 1$ 3.12: if $\text{untested}[t] > 0$ then Skip t 3.13: if $B \xrightarrow{T} \not\subseteq R$ then 3.14: for all non-inert $t \xrightarrow{\alpha} u$ do 3.15: if $t \xrightarrow{\alpha} u \in T$ then Skip t 3.16: end for 3.17: end if 3.18: Add t to U 3.19: end for 3.20: end for 3.21: if $U > \frac{1}{2} B$ then 3.22: Abort this coroutine 3.23: end if 3.24: Abort the other coroutine 3.25: return $(B \setminus U, U)$ 3.26: end coroutines </pre>	<pre> $R := R \cup B \xrightarrow{T \setminus \text{Marked}(T)}$ for all $s \in R$ while $R \leq \frac{1}{2} B$ do for all inert $t \xrightarrow{\tau} s$ do Add t to R end for end for if $R > \frac{1}{2} B$ then Abort this coroutine end if return $(R, B \setminus R)$ </pre>	<pre> $\} O(\text{Marked}(T))$ $\} O(1)$ or $O(R_{\rightarrow})$ $\} O(U_{\leftarrow})$ or $O(R_{\leftarrow})$ $\} O(U_{\rightarrow} + (\text{Bottom}(R) \setminus \text{Bottom}(B))_{\rightarrow})$ $\} O(U_{\leftarrow})$ or $O(R_{\leftarrow})$ $\} O(1)$ </pre>
--	--	--

To identify the states in U , observe that a state is in U if all its inert successors are in U and it does not have a transition in $T_{B \rightarrow}$. To compute U , we let a counter $untested[t]$ for every non-bottom state t record the number of outgoing inert transitions to states that are not yet known to be in U . If $untested[t] = 0$, this means all inert successors of t are guaranteed to be in U , so, provided t does not have a transition in $T_{B \rightarrow}$, one can also add t to U . To take care of the possibility that all inert transitions of t have been visited before all sources of unmarked transitions in $T_{B \rightarrow}$ are added to R , we check all non-inert transitions of t to determine whether they are not in $T_{B \rightarrow}$ at Lines 3.13ℓ–3.17ℓ.

The coroutine that finishes first, provided that its number of states does not exceed $\frac{1}{2}|B|$, has completely computed the smaller subblock resulting from the refinement, and the other coroutine can be aborted. As soon as the number of states of a coroutine is known to exceed $\frac{1}{2}|B|$, it is aborted, and the other coroutine can continue to identify the smaller subblock. In detail, the runtime complexity of $\langle R, U \rangle := \text{split}(B, T)$ is:

- $O(|R_{\rightarrow}| + |R_{\leftarrow}|)$, if $|R| \leq |U|$, and
- $O(|\text{Marked}(T)| + |U_{\rightarrow}| + |U_{\leftarrow}| + |(Bottom(R) \setminus Bottom(B))_{\rightarrow}|)$, if $|U| \leq |R|$.

This complexity is inferred as follows. As we execute the coroutines in lockstep, it suffices to show that the runtime bound for the smaller subblock is satisfied.

In case $|R| \leq |U|$, observe $|\text{Marked}(T)| \leq |R_{\rightarrow}|$, so we get $O(|R_{\rightarrow}| + |R_{\leftarrow}|)$ directly from the R -coroutine. When $|U| \leq |R|$, we use time in $O(|\text{Marked}(T)|)$ for Line 3.2, and we use time in $O(|U_{\leftarrow}|)$ for everything else except Lines 3.13ℓ–3.17ℓ. For these latter lines, we distinguish two cases. If it turns out that t has no transition $t \xrightarrow{\alpha} u \in T$, it is a U -state, so we attribute the time to $O(|U_{\rightarrow}|)$. Otherwise, it is an R -state that had some inert transitions in B , but they all are now in $R \xrightarrow{\tau} U$. So t is a new bottom state, and we attribute the time to the outgoing transitions of new bottom states: $O(|(Bottom(R) \setminus Bottom(B))_{\rightarrow}|)$.

5 Experimental evaluation

The new algorithm (JGKW20) has been implemented in the mCRL2 toolset [5] and is available in its 201908.0 release. This toolset also contains implementations of various other algorithms, such as the $O(mn)$ algorithm by Groote and Vaandrager (GV) [10] and the $O(m(\log |Act| + \log n))$ algorithm of [9] (GJKW17). In addition, it offers a sequential implementation of the partition-refinement algorithm using state signatures by Blom and Orzan (BO) [3], which has time complexity $O(n^2m)$. For each state, BO maintains a signature describing which blocks the state can reach directly via its outgoing transitions.

In this section, we report on the experiments we have conducted to compare GV, BO, GJKW17 and JGKW20 when applied to practical examples. In the experiments the given LTSs are minimised w.r.t. branching bisimilarity. The set of benchmarks consists of all LTSs offered by the VLTS benchmark set³ with at least 60,000 transitions. Their name ends in “-[$n/1000$]-[$m/1000$]” and thus

³ <http://cadp.inria.fr/resources/vlts>.

Table 1: Running time and memory use results. ▼ and ▲: significantly better (worse) than the others.

model	time			space				
	GV	BO	GJKW17	JGKW20	GV	BO	GJKW17	JGKW20
vasy_40.60	24. s	138. s ▲	.1 s	.05 s ▼	65.5 MB	60.6 MB	70 MB	60 MB
vasy_18.73	.21 s	.37 s ▲	.11 s	.07 s ▼	55.6 MB	56.7 MB	50 MB	50 MB
vasy_157.297	1.7 s	2. s	.4 s	.2 s ▼	97.3 MB	94.3 MB	127.2 MB ▲	90 MB
vasy_52.318	.31 s	.9 s ▲	.2 s	.2 s	73.4 MB	90.4 MB	90.6 MB ▲	73.4 MB
vasy_83.325	2.6 s ▲	1.0 s	.9 s	.3 s ▼	116.2 MB	.11 GB	230.5 MB ▲	.10 GB
vasy_116.368	.9 s	5. s ▲	.6 s	4 s ▼	92.8 MB	110.6 MB	.13 GB ▲	90 MB
vasy_720.390	.4 s	.9 s	.6 s	.4 s	105.2 MB	103.2 MB	.19 GB ▲	95.9 MB ▼
vasy_69.520	1.5 s	4. s ▲	2.4 s	.8 s ▼	.15 GB	.15 GB	358.1 MB ▲	162.0 MB
cwi_371.641	7.4 s ▲	5.9 s	1. s	.7 s	.17 GB	229.0 MB ▲	185.4 MB	.14 GB ▼
vasy_166.651	4.9 s ▲	1.9 s	2. s	.7 s ▼	157.5 MB	141.8 MB	342.9 MB ▲	139.5 MB ▼
cwi_214.684	1.4 s	9. s ▲	.5 s	.5 s	140.7 MB	162.1 MB ▲	152.0 MB	.13 GB
cwi_142.925	1.4 s ▲	.8 s	1.0 s	.9 s	152.5 MB	117.9 MB ▼	156.6 MB ▲	152.5 MB
vasy_386.1171	1.4 s	2. s ▲	1.3 s	.9 s ▼	229.2 MB	210.1 MB ▼	273.4 MB ▲	228.7 MB
vasy_66.1302	3. s	4.7 s	5. s	2.2 s ▼	.23 GB ▼	283.1 MB	618.1 MB ▲	268.0 MB
vasy_164.1619	2.0 s	5. s ▲	3. s	-	.25 GB	235.4 MB	262.4 MB	245.0 MB
vasy_65.2621	90 s ▲	11. s	20 s	4.7 s ▼	.5 GB	534.7 MB	1.8 GB	.5 GB
cwi_566.3984	8. s	7. s	8. s	6. s	.5 GB	351.5 MB ▼	514.0 MB ▲	.5 GB
vasy_1112.5290	10. s	17. s ▲	10 s	10 s	.8 GB	720.9 MB	931.5 MB	.7 GB
cwi_2165.8723	.4 min	3. min ▲	-	.3 min	1.3 GB	1.8726 GB	2.1321 GB ▲	1.2 GB
vasy_6120.11031	2. min ▲	1.7 min	-	.4 min	1.8 GB	1.7379 GB	3.6596 GB ▲	1.5960 GB ▼
cwi_2416.17605	10 min ▲	3. min	-	-	1.5999 GB	1.7434 GB	4.1299 GB ▲	1.4 GB ▼
vasy_574.13561	50 s	56. s	-	-	1.8835 GB ▲	1.5217 GB	1.5 GB	1.5 GB
vasy_4230.13944	30 min ▲	5. min	-	.6 min	2.0965 GB	2.3188 GB	5.8661 GB ▲	2.0 GB ▼
vasy_4338.15666	34. min ▲	3. min	2. min	.8 min ▼	2.4043 GB	2.3559 GB	5.9888 GB ▲	1.8535 GB ▼
vasy_6020.19353	30 s	19. s	20 s	20 s	1.6 GB	1.5157 GB ▼	1.6638 GB	1.6748 GB ▲
vasy_11026.24660	50 min ▲	20 min	6. s	5. s	870. MB	2.3442 GB ▲	870. MB	870. MB
lift6-final	1.0 min	3. min ▲	-	1. min ▼	3.6475 GB	4.0513 GB	9.6425 GB ▲	3.4412 GB ▼
vasy_12923.27667	40 min ▲	10 min	-	.9 min	3.3846 GB	8.1984 GB ▲	6.2971 GB	3.2125 GB ▼
vasy_8082.42933	2. min	5. min ▲	-	1. min	4.0091 GB	4.5371 GB	10.6743 GB ▲	3.7298 GB ▼
cwi_7838.59101	5. min	100 min ▲	6. min	3. min	6.5283 GB ▼	8.3266 GB	13.7899 GB ▲	6.7646 GB
dining.14	17. min	20 min	20 min	10 min	20.4826 GB ▼	21.7156 GB	23.7810 GB ▲	20.9756 GB
cwi_33949.165318	11. min	80 min ▲	20 min	8. min	22.7304 GB	33.0351 GB	37.8606 GB ▲	21.0611 GB ▼
I394-fn3	25. h	▲ 3. h	.5 h	.3 h	▼ 37.4893 GB	71.8698 GB	53.2166 GB	31.5132 GB ▼
Total	28. h	▲ 8. h	1.4 h	.8 h	▼ 121.8 GB	176.33 GB	194.2 GB	▲ 112.0 GB ▼

describes their size. Additionally, we consider three cases that have been derived from models distributed with the mCRL2 toolset:

1. **lift6-final**: this model is based on an elevator model, extended to six elevators ($n = 6,047,527$, $m = 26,539,368$);
2. **dining_14**: this is the dining philosophers model with 14 philosophers ($n = 18,378,370$, $m = 164,329,284$);
3. **1394-fin3**: this is an altered version of the 1394-fin model, extended to three processes and two data elements ($n = 126,713,623$, $m = 276,426,688$).

The software and benchmarks used for the experiments are available online [15]. All experiments have been conducted on individual nodes of the DAS-5 cluster [1]. Each of these nodes was running CENTOS LINUX 7.4, had an INTEL XEON E5-2698-v3 2.3GHz CPU, and was equipped with 256 GB RAM. Development version 201808.0.c59cfd413f of mCRL2 was used for the experiments.⁴

Table 1 presents the obtained results. Benchmarks are ordered by their number of transitions. On each benchmark, we have applied each algorithm ten times, and report the mean runtime and memory use of these ten runs, rounded to significant digits (estimated using [4] for the standard deviation). A trailing decimal dot indicates that the unit digit is significant. If this dot is missing, there is one insignificant zero. For all presented data the estimated standard deviation is less than 20% of the mean. Otherwise we print ‘-’ in Table 1.

The ▼-symbol after a table entry indicates that the measurement is significantly better than the corresponding measurements for the other three algorithms, and the ▲-symbol indicates that it is significantly worse. Here, the results are considered significant if, given a hundred tables such as Table 1, one table of running time (resp. memory) is expected to contain spuriously significant results.

Concerning the runtimes, clearly, GV and BO perform significantly worse than the other two algorithms, and JGKW20 in many cases performs significantly better than the others. In particular, JGKW20 is about 40% faster than GJKW17, the fastest older algorithm. Concerning memory use, in the majority of cases GJKW17 uses more memory than the others, while sometimes BO is the most memory-hungry. JGKW20 is much more competitive, in many cases even outperforming every other algorithm.

The results show that when applied to practical cases, JGKW20 is generally the fastest algorithm, and even when other algorithms have similar runtimes, it uses almost always the least memory. This combination makes JGKW20 currently the best option for branching bisimulation minimisation of LTSs.

Data Availability Statement and Acknowledgement. The datasets generated and analysed during the current study are available in the figshare repository: <https://doi.org/10.6084/m9.figshare.11876688.v1>. This work is partly done during a visit of the first author at Eindhoven University of Technology, and a visit of the second author at the Institute of Software, Chinese Academy of Sciences. The first author is supported by the National Natural Science Foundation of China, Grant No. 61761136011.

⁴ <https://github.com/mCRL2org/mCRL2/commit/c59cfd413f>

References

1. Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., Wijshoff, H.: A medium-scale distributed system for computer science research: Infrastructure for the long term. *IEEE Computer* **49**(5), 54–63 (2016). <https://doi.org/10.1109/MC.2016.127>
2. Bartholomeus, M., Luttkik, B., Willemse, T.: Modelling and analysing ERTMS Hybrid Level 3 with the mCRL2 toolset. In: Howar, F., Barnat, J. (eds.) *Formal methods for industrial critical systems: FMICS*. LNCS, vol. 11119, pp. 98–114. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00244-2_7
3. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. *Electron. Notes Theor. Comput. Sci.* **80**(1), 99–113 (2003). [https://doi.org/10.1016/S1571-0661\(05\)80099-4](https://doi.org/10.1016/S1571-0661(05)80099-4)
4. Bruggen, R.M.: A note on unbiased estimation of the standard deviation. *The American Statistician* **23**(4), 32 (1969). <https://doi.org/10.1080/00031305.1969.10481865>
5. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A.J., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems. In: Vojnar, T., Zhang, L. (eds.) *Tools and algorithms for the construction and analysis of systems: TACAS, Part II*. LNCS, vol. 11428, pp. 21–39. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_2
6. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. *J. ACM* **42**(2), 458–487 (1995). <https://doi.org/10.1145/201019.201032>
7. van Glabbeek, R.J.: The linear time – branching time spectrum II. In: Best, E. (ed.) *CONCUR'93: 4th international conference on concurrency theory*. LNCS, vol. 715, pp. 66–81. Springer, Berlin (1993). https://doi.org/10.1007/3-540-57208-2_6
8. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. *J. ACM* **43**(3), 555–600 (1996). <https://doi.org/10.1145/233551.233556>
9. Groote, J.F., Jansen, D.N., Keiren, J.J.A., Wijs, A.J.: An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. *ACM Trans. Comput. Logic* **18**(2), Article 13 (2017). <https://doi.org/10.1145/3060140>
10. Groote, J.F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Paterson, M.S. (ed.) *Automata, languages and programming [ICALP]*, LNCS, vol. 443, pp. 626–638. Springer, Berlin (1990). <https://doi.org/10.1007/BFb0032063>
11. Groote, J.F., Wijs, A.J.: An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In: Chechik, M., Raskin, J.F. (eds.) *Tools and algorithms for the construction and analysis of systems: TACAS*. LNCS, vol. 9636, pp. 607–624. Springer, Berlin (2016). https://doi.org/10.1007/978-3-662-49674-9_40
12. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: 36th annual symposium on foundations of computer science [FOCS]. pp. 453–462. IEEE Comp. Soc., Los Alamitos, Calif. (1995). <https://doi.org/10.1109/SFCS.1995.492576>
13. Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) *Theory of machines and computations*, pp. 189–196. Academic Press, New York (1971). <https://doi.org/10.1016/B978-0-12-417750-5.50022-1>
14. Jansen, D.N., Groote, J.F., Keiren, J.J.A., Wijs, A.J.: A simpler $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. arXiv preprint 1909.10824 (2019), <https://arxiv.org/abs/1909.10824>

15. Jansen, D.N., Groote, J.F., Keiren, J.J.A., Wijs, A.J.: An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. Figshare (2020), <https://doi.org/10.6084/m9.figshare.11876688.v1>
16. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.* **86**(1), 43–68 (1990). [https://doi.org/10.1016/0890-5401\(90\)90025-D](https://doi.org/10.1016/0890-5401(90)90025-D)
17. Milner, R.: A calculus of communicating systems, LNCS, vol. 92. Springer, Berlin (1980). <https://doi.org/10.1007/3-540-10235-3>
18. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. *SIAM J. Comput.* **16**(6), 973–989 (1987). <https://doi.org/10.1137/0216062>
19. Reniers, M.A., Schoren, R., Willemse, T.A.C.: Results on embeddings between state-based and event-based systems. *Comput. J.* **57**(1), 73–92 (2014). <https://doi.org/10.1093/comjnl/bxs156>
20. Valmari, A.: Bisimilarity minimization in $O(m \log n)$ time. In: Franceschinis, G., Wolf, K. (eds.) *Applications and theory of Petri nets: PETRI NETS*, LNCS, vol. 5606, pp. 123–142. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-02424-5_9
21. Valmari, A., Lehtinen, P.: Efficient minimization of DFAs with partial transition functions. In: Albers, S., Weil, P. (eds.) *25th international symposium on theoretical aspects of computer science: STACS, LIPIcs*, vol. 1, pp. 645–656. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2008). <https://doi.org/10.4230/LIPIcs.STACS.2008.1328>

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

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





Verifying Quantum Communication Protocols with Ground Bisimulation^{*}

Xudong Qin^{1,2}, Yuxin Deng¹ , and Wenjie Du³



¹ Shanghai Key Laboratory of Trustworthy Computing, MOE International Joint Lab of Trustworthy Software, and International Research Center of Trustworthy Software, East China Normal University, Shanghai, China
steven.qxd@126.com yxdeng@sei.ecnu.edu.cn

² Peng Cheng Laboratory, Shenzhen, China

³ Shanghai Normal University, Shanghai, China
wenjiedu@shnu.edu.cn

Abstract. One important application of quantum process algebras is to formally verify quantum communication protocols. With a suitable notion of behavioural equivalence and a decision method, one can determine if an implementation of a protocol is consistent with its specification. Ground bisimulation is a convenient behavioural equivalence for quantum processes because of its associated coinduction proof technique. We exploit this technique to design and implement two on-the-fly algorithms for the strong and weak versions of ground bisimulation to check if two given processes in quantum CCS are equivalent. We then develop a tool that can verify interesting quantum protocols such as the BB84 quantum key distribution scheme.

Keywords: Quantum process algebra · Bisimulation · Verification · Quantum communication protocols.

1 Introduction

Process algebras provide a useful formal method for specifying and verifying concurrent systems. Their extensions to the quantum setting have also appeared in the literature. For example, Jorrand and Lalire [18,21] defined the *Quantum Process Algebra* (QPAI) and presented a branching bisimulation to identify quantum processes with the same branching structure. Gay and Nagarajan [15] developed *Communicating Quantum Processes* (CQP), for which Davidson [6] established a bisimulation congruence. Feng et al. [10] have proposed a quantum variant of Milner's CCS [23], called qCCS, and a notion of probabilistic bisimulation for quantum processes, which is then improved to be a general notion of bisimulation that enjoys a congruence property [12]. Later on, motivated by [25], Deng and Feng [9] defined an open bisimulation for quantum processes

^{*} Supported by the National Natural Science Foundation of China (61672229, 61832015) and the Inria-CAS joint project Quasar.

that makes it possible to separate ground bisimulation and the closedness under super-operator applications, thus providing not only a neater and simpler definition, but also a new technique for proving bisimilarity. In order to avoid the problem of instantiating quantum variables by potentially infinitely many quantum states, Feng et al. [11] extended the idea of symbolic bisimulation [17] for value-passing CCS and provided a symbolic version of open bisimulation for qCCS. They proposed an algorithm for checking symbolic ground bisimulation.

In the current work, we consider the ground bisimulation proposed in [9]. We put forward an on-the-fly algorithm to check if two given processes in qCCS with fixed initial quantum states are ground bisimilar. The algorithm is simpler than the one in [11] because the initial quantum states are determined for the former but can be parametric for the latter. Moreover, in many applications, we are only interested in the correctness of a quantum protocol with a predetermined input of quantum states. This is especially the case in the design stage of a protocol or in the debugging of a program.

The ground bisimulation defined in [9] is a notion of weak bisimulation because a strong transition can be matched by a weak transition where invisible actions are abstracted away. We also consider a strong version where all actions are visible, for which we have a simpler algorithm. Both algorithms are obtained by adapting the on-the-fly algorithm for checking probabilistic bisimulations [8,7], which in turn has its root in similar algorithms for checking classical bisimulations [14,17]. The basic idea is as follows. A quantum process with an initial quantum state forms a configuration. We describe the operational behaviour of a configuration as a probabilistic labelled transition system (pLTS), where probabilistic transitions arise naturally because measuring a quantum system can entail a probability distribution of post-measurement quantum systems. Ground bisimulations are a strengthening of probabilistic bisimulations by imposing some constraints on quantum variables and the environment states of processes. The skeleton of the algorithm for the strong ground bisimulation resembles to that for strong probabilistic bisimulation [8]. The algorithm for the (weak) ground bisimulation is inspired by [28] and uses as a subroutine a procedure in the aforementioned work. The procedure reduces the problem of finding a matching weak transition to a linear programming problem that can be solved in polynomial time. We have developed a tool that implements both algorithms and can check if two given configurations are strongly or weakly bisimilar. It is useful to validate whether an implementation of a protocol is equivalent to the specification. We have conducted experiments on a few interesting quantum protocols including super-dense coding, teleportation, secret sharing, and several quantum key distribution protocols, in particular the BB84 protocol [5], to analyse the functional correctness of the protocols.

Other related work Ardeshir-Larijani et al. [3] proposed a quantum variant of CCS to describe quantum protocols. The syntax of that variant is similar to qCCS but its semantics is very different. The behaviour of a concurrent process is a finite tree and an interleaving is a path from the root to a leaf. By interpreting an interleaving as a superoperator [26], the semantics of a process

is a set of superoperators. The equivalence checking between two processes boils down to the equivalence checking between superoperators, which is accomplished by using the stabiliser simulation algorithm invented by Aaronson and Gottesman [1]. Ardeshir-Larijani et al. have implemented their approach in an equivalence checker in Java and verified several quantum protocols from teleportation to secret sharing. However, they are not able to handle the BB84 quantum key distribution protocol because its correctness cannot be specified as an equivalence between interleavings. Our approach is based on ground bisimulation and keeps all the branching behaviour of a concurrent process. Our algorithms for checking ground bisimulations are influenced by the on-the-fly algorithm of Hennessey and Lin for value-passing CCS [17]. We are inspired by the probabilistic bisimulation checking algorithm of Baier et al. [4] for the strong version of ground bisimulation, and by the weak bisimulation checking algorithm of Turrini and Hermanns [28] for the weak version.

Kubota et al. [20] implemented a semi-automated tool to check a notion of symbolic bisimulation and used it to verify the equivalence of BB84 and another quantum key distribution protocol based on entanglement distillation [27]. There are two main differences between their work and ours. (1) Their tool is based on equational reasoning and thus requires a user to provide equations while our tool is fully automatic. (2) Their semantic interpretation of measurement is different and entails a kind of linear-time semantics for quantum processes that ignores the timepoints of the occurrences of probabilistic branches. However, we use a branching-time semantics. For instance, the occurrence of a measurement before or after a visible action is significant for our semantics but not for the semantics proposed in [20].

Besides equivalence checking, based on either superoperators or bisimulations as mentioned above, model checking is another feasible approach to verify quantum protocols. For instance, Gay et al. developed the QMC model checker [16]. Feng et al. implemented the tool QPMC [13] to model check quantum programs and protocols. There are also other approaches for verifying quantum systems. Abramsky and Coecke [2] proposed a categorical semantics for quantum protocols. Quantomatic [19] is a semi-automated tool based on graph rewriting. Ying [30] established a quantum Hoare logic, which has been implemented in a theorem prover [22].

The rest of the paper is structured as follows. In Section 2 we recall the syntax and semantics of the quantum process algebra qCCS. In Section 3 we present an algorithm for checking ground bisimulations. In Section 4 we report the implementation of the algorithm and some experimental results on verifying a few quantum communication protocols. Finally, we conclude in Section 5 and discuss some future work.

2 Quantum CCS

We introduce a quantum extension of classical CCS (qCCS) which was originally studied in [10,29,12]. Three types of data are considered in qCCS: as classical

$$\begin{array}{ll}
qv(\mathbf{nil}) = \emptyset & qv(\tau.P) = qv(P) \\
qv(c?x.P) = qv(P) & qv(c!e.P) = qv(P) \\
qv(\underline{c}?q.P) = qv(P) - \{q\} & qv(\underline{c}!q.P) = qv(P) \cup \{q\} \\
qv(\mathcal{E}[\tilde{q}].P) = qv(P) \cup \tilde{q} & qv(M[\tilde{q}; x].P) = qv(P) \cup \tilde{q} \\
qv(P + Q) = qv(P) \cup qv(Q) & qv(P \parallel Q) = qv(P) \cup qv(Q) \\
qv(P[f]) = qv(P) & qv(P \setminus L) = qv(P) \\
qv(\mathbf{if } b \mathbf{ then } P) = qv(P) & qv(A(\tilde{q}; \tilde{x})) = \tilde{q}.
\end{array}$$

Fig. 1. Free quantum variables

data we have **Bool** for booleans and **Real** for real numbers, and as quantum data we have **Qbt** for qubits. Consequently, two countably infinite sets of variables are assumed: $cVar$ for classical variables, ranged over by x, y, \dots , and $qVar$ for quantum variables, ranged over by q, r, \dots . We assume a set Exp , which includes $cVar$ as a subset and is ranged over by e, e', \dots , of classical data expressions over **Real**, and a set of boolean-valued expressions $BExp$, ranged over by b, b', \dots , with the usual boolean constants **true**, **false**, and operators \neg, \wedge, \vee , and \rightarrow . In particular, we let $e \bowtie e'$ be a boolean expression for any $e, e' \in Exp$ and $\bowtie \in \{>, <, \geq, \leq, =\}$. We further assume that only classical variables can occur freely in both data expressions and boolean expressions. Two types of channels are used: $cChan$ for classical channels, ranged over by c, d, \dots , and $qChan$ for quantum channels, ranged over by $\underline{c}, \underline{d}, \dots$. A relabelling function f is a map on $cChan \cup qChan$ such that $f(cChan) \subseteq cChan$ and $f(qChan) \subseteq qChan$. Sometimes we abbreviate a sequence of distinct variables q_1, \dots, q_n into \tilde{q} .

The terms in qCCS are given by:

$$\begin{array}{l}
P, Q ::= \mathbf{nil} \mid \tau.P \mid c?x.P \mid c!e.P \mid \underline{c}?q.P \mid \underline{c}!q.P \mid \mathcal{E}[\tilde{q}].P \mid M[\tilde{q}; x].P \mid \\
P + Q \mid P \parallel Q \mid P[f] \mid P \setminus L \mid \mathbf{if } b \mathbf{ then } P \mid A(\tilde{q}; \tilde{x})
\end{array}$$

where f is a relabelling function and $L \subseteq cChan \cup qChan$ is a set of channels. Most of the constructors are standard as in CCS [23]. We briefly explain a few new constructors. The process $\underline{c}?q.P$ receives a quantum datum along quantum channel \underline{c} and evolves into P , while $\underline{c}!q.P$ sends out a quantum datum along quantum channel \underline{c} before evolving into P . The symbol \mathcal{E} represents a trace-preserving super-operator applied on the quantum system referred to by the variables \tilde{q} . The process $M[\tilde{q}; x].P$ measures the state of qubits \tilde{q} according to the observable M and stores the measurement outcome into the classical variable x of P .

Free classical variables can be defined in the usual way, except for the fact that the variable x in the quantum measurement $M[\tilde{q}; x]$ is bound. A process P is closed if it contains no free classical variable, i.e. $fv(P) = \emptyset$.

The set of free quantum variables for process P , denoted by $qv(P)$ can be inductively defined as in Figure 1. For a process to be legal, we require that

1. $q \notin qv(P)$ in the process $\underline{c}!q.P$;
2. $qv(P) \cap qv(Q) = \emptyset$ in the process $P \parallel Q$;

<p>(<i>Tau</i>) $\langle \tau.P, \rho \rangle \xrightarrow{\tau} \langle P, \rho \rangle$</p> <p>(<i>C-Outp</i>) $v = \llbracket e \rrbracket$ $\langle c!e.P, \rho \rangle \xrightarrow{c!v} \langle P, \rho \rangle$</p> <p>(<i>Q-inp</i>) $r \notin qv(c?q.P)$ $\langle c?q.P, \rho \rangle \xrightarrow{c?r} \langle P[r/q], \rho \rangle$</p> <p>(<i>Q-Com</i>) $\langle P_1, \rho \rangle \xrightarrow{c?r} \langle P'_1, \rho \rangle \quad \langle P_2, \rho \rangle \xrightarrow{c!r} \langle P'_2, \rho \rangle$ $\langle P_1 \parallel P_2, \rho \rangle \xrightarrow{\tau} \langle P'_1 \parallel P'_2, \rho \rangle$</p> <p>(<i>Meas</i>) $M = \sum_{i \in I} \lambda_i E^i \quad p_i = \text{tr}(E_{\tilde{q}}^i \rho)$ $\langle M[\tilde{q}; x].P, \rho \rangle \xrightarrow{\tau} \sum_{i \in I} p_i \langle P[\lambda_i/x], E_{\tilde{q}}^i \rho E_{\tilde{q}}^i / p_i \rangle$</p> <p>(<i>Int</i>) $\langle P_1, \rho \rangle \xrightarrow{\alpha} \Delta \quad qbv(\alpha) \cap qv(P_2) = \emptyset$ $\langle P_1 \parallel P_2, \rho \rangle \xrightarrow{\alpha} \Delta \parallel P_2$</p> <p>(<i>Rel</i>) $\langle P, \rho \rangle \xrightarrow{\alpha} \Delta$ $\langle P[f], \rho \rangle \xrightarrow{f(\alpha)} \Delta[f]$</p> <p>(<i>Cho</i>) $\langle P, \rho \rangle \xrightarrow{\alpha} \Delta \quad \llbracket b \rrbracket = \text{true}$ $\langle \text{if } b \text{ then } P, \rho \rangle \xrightarrow{\alpha} \Delta$</p>	<p>(<i>C-Inp</i>) $v \in \text{Real}$ $\langle c?x.P, \rho \rangle \xrightarrow{c?v} \langle P[v/x], \rho \rangle$</p> <p>(<i>C-Com</i>) $\langle P_1, \rho \rangle \xrightarrow{c?v} \langle P'_1, \rho \rangle \quad \langle P_2, \rho \rangle \xrightarrow{c!v} \langle P'_2, \rho \rangle$ $\langle P_1 \parallel P_2, \rho \rangle \xrightarrow{\tau} \langle P'_1 \parallel P'_2, \rho \rangle$</p> <p>(<i>Q-Outp</i>) $\langle c!q.P, \rho \rangle \xrightarrow{c!q} \langle P, \rho \rangle$</p> <p>(<i>Oper</i>) $\langle \mathcal{E}[\tilde{q}].P, \rho \rangle \xrightarrow{\tau} \langle P, \mathcal{E}_{\tilde{q}}(\rho) \rangle$</p> <p>(<i>Sum</i>) $\langle P_1, \rho \rangle \xrightarrow{\alpha} \Delta$ $\langle P_1 + P_2, \rho \rangle \xrightarrow{\alpha} \Delta$</p> <p>(<i>Res</i>) $\langle P, \rho \rangle \xrightarrow{\alpha} \Delta \quad \text{cn}(\alpha) \cap L = \emptyset$ $\langle P \setminus L, \rho \rangle \xrightarrow{\alpha} \Delta \setminus L$</p> <p>(<i>Cons</i>) $\langle P[\tilde{v}/\tilde{x}, \tilde{r}/\tilde{q}], \rho \rangle \xrightarrow{\alpha} \Delta \quad A(\tilde{x}, \tilde{q}) := P$ $\langle A(\tilde{v}, \tilde{r}), \rho \rangle \xrightarrow{\alpha} \Delta$</p>
---	---

Fig. 2. Operational semantics of qCCS. Here in rule (*C-Outp*), $\llbracket e \rrbracket$ is the evaluation of e , and in rule (*Meas*), $E_{\tilde{q}}^i$ denotes the operator E^i acting on the quantum systems \tilde{q} .

3. Each constant $A(\tilde{q}; \tilde{x})$ has a defining equation $A(\tilde{q}; \tilde{x}) := P$, where P is a term with $qv(P) \subseteq \tilde{q}$ and $fv(P) \subseteq \tilde{x}$.

The first condition says that a quantum system will not be referenced after it has been sent out. This is a requirement of the quantum no-cloning theorem. The second condition says that parallel composition \parallel models separate parties that never reference a quantum system simultaneously.

Throughout the paper we implicitly assume the convention that processes are identified up to α -conversion, bound variables differ from each other and they are different from free variables.

Before introducing the operational semantics of qCCS processes, we review the model of probabilistic labelled transition systems (pLTSs). Later on we will interpret the behaviour of quantum processes in terms of pLTSs because quantum measurements give rise to probability distributions naturally.

We begin with some notations. A (discrete) probability distribution over a set S is a function $\Delta: S \rightarrow [0, 1]$ with $\sum_{s \in S} \Delta(s) = 1$; the support of such a Δ is the set $[\Delta] = \{s \in S \mid \Delta(s) > 0\}$. The point distribution \bar{s} assigns probability 1 to s and 0 to all other elements of S , so that $[\bar{s}] = \{s\}$. We only need to use distributions with finite supports, and let $Dist(S)$ denote the set of finite support distributions over S , ranged over by Δ, Θ , etc. If $\sum_{k \in K} p_k = 1$ for some collection of $p_k \geq 0$, and the Δ_k are distributions, then so is $\sum_{k \in K} p_k \cdot \Delta_k$ with $(\sum_{k \in K} p_k \cdot \Delta_k)(s) = \sum_{k \in K} p_k \cdot \Delta_k(s)$.

Definition 1. A probabilistic labelled transition system is a triple $\langle S, Act_\tau, \rightarrow \rangle$, where S is a set of states, Act_τ is a set of visible actions Act augmented with the invisible action τ , and $\rightarrow \subseteq S \times Act_\tau \times Dist(S)$ is the transition relation.

We often write $s \xrightarrow{\alpha} \Delta$ for $(s, \alpha, \Delta) \in \rightarrow$. In pLTSs we not only consider relations between states, but also relations between distributions. Therefore, we make use of the lifting operation below [7].

Definition 2. Let $\mathcal{R} \subseteq S \times S$ be a relation between states. Then $\mathcal{R}^\circ \subseteq Dist(S) \times Dist(S)$ is the smallest relation that satisfies the two rules: (i) $s \mathcal{R} s'$ implies $\bar{s} \mathcal{R}^\circ \bar{s}'$; (ii) $\Delta_i \mathcal{R}^\circ \Theta_i$ for all $i \in I$ implies $(\sum_{i \in I} p_i \cdot \Delta_i) \mathcal{R}^\circ (\sum_{i \in I} p_i \cdot \Theta_i)$ for any $p_i \in [0, 1]$ with $\sum_{i \in I} p_i = 1$, where I is a finite index set.

We apply this operation to the relations $\xrightarrow{\alpha}$ in the pLTS for $\alpha \in Act_\tau$, where we also write $\xrightarrow{\alpha}$ for $(\xrightarrow{\alpha})^\circ$. Thus as source of a relation $\xrightarrow{\alpha}$ we now also allow distributions. But note that $\bar{s} \xrightarrow{\alpha} \Delta$ is more general than $s \xrightarrow{\alpha} \Delta$ because if $\bar{s} \xrightarrow{\alpha} \Delta$ then there is a collection of distributions Δ_i and probabilities p_i such that $s \xrightarrow{\alpha} \Delta_i$ for each $i \in I$ and $\Delta = \sum_{i \in I} p_i \cdot \Delta_i$ with $\sum_{i \in I} p_i = 1$.

We write $s \xrightarrow{\hat{\tau}} \Delta$ if either $s \xrightarrow{\tau} \Delta$ or $\Delta = \bar{s}$. We define weak transitions $\xrightarrow{\hat{a}}$ by letting $\xrightarrow{\hat{\tau}}$ be the reflexive and transitive closure of $\xrightarrow{\hat{\tau}}$ and writing $\Delta \xrightarrow{\hat{a}} \Theta$ for $a \in Act$ whenever $\Delta \xrightarrow{\hat{\tau}} \xrightarrow{a} \xrightarrow{\hat{\tau}} \Theta$. If $\Delta = \bar{s}$ is a point distribution, we often write $s \xrightarrow{\hat{a}} \Theta$ instead of $\bar{s} \xrightarrow{\hat{a}} \Theta$.

We now give the semantics of qCCS. For each quantum variable q we assume a 2-dimensional Hilbert space \mathcal{H}_q . For any nonempty subset $S \subseteq qVar$ we write \mathcal{H}_S for the tensor product space $\bigotimes_{q \in S} \mathcal{H}_q$ and $\mathcal{H}_{\bar{S}}$ for $\bigotimes_{q \notin S} \mathcal{H}_q$. In particular, $\mathcal{H} = \mathcal{H}_{qVar}$ is the state space of the whole environment consisting of all the quantum variables, which is a countably infinite dimensional Hilbert space.

Let P be a closed quantum process and ρ a density operator on \mathcal{H}^1 , the pair $\langle P, \rho \rangle$ is called a *configuration*. We write Con for the set of all configurations, ranged over by \mathcal{C} and \mathcal{D} . We interpret qCCS with a pLTS whose states are all the configurations definable in the language, and whose transitions are determined by the rules in Figure 2; we have omitted the obvious symmetric counterparts to the rules (*C-Com*), (*Q-Com*), (*Int*) and (*Sum*). The set of actions Act takes the following form, consisting of classical/quantum input/output actions.

$$Act = \{c?v, c!v \mid c \in cChan, v \in \mathbf{Real}\} \cup \{\underline{c}?r, \underline{c}!r \mid \underline{c} \in qChan, r \in qVar\}$$

¹ As \mathcal{H} is infinite dimensional, ρ should be understood as a density operator on some finite dimensional subspace of \mathcal{H} which contains $\mathcal{H}_{qv(P)}$.

We use $cn(\alpha)$ for the set of channel names in action α . For example, we have $cn(\underline{c}?x) = \{\underline{c}\}$ and $cn(\tau) = \emptyset$.

In the first eight rules in Figure 2, the targets of arrows are point distributions, and we use the slightly abbreviated form $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ to mean $\mathcal{C} \xrightarrow{\alpha} \overline{\mathcal{C}'}$.

The rules use the obvious extension of the function \parallel on terms to configurations and distributions. To be precise, $\mathcal{C} \parallel P$ is the configuration $\langle Q \parallel P, \rho \rangle$ where $\mathcal{C} = \langle Q, \rho \rangle$, and $\Delta \parallel P$ is the distribution defined by:

$$(\Delta \parallel P)(\langle Q, \rho \rangle) \stackrel{def}{=} \begin{cases} \Delta(\langle Q', \rho \rangle) & \text{if } Q = Q' \parallel P \text{ for some } Q' \\ 0 & \text{otherwise.} \end{cases}$$

Similar extension applies to $\Delta[f]$ and $\Delta \setminus L$.

Suppose there is a configuration $\mathcal{C} = \langle P, \rho \rangle$, the partial trace over system P at such state can be defined as $tr_{qv(P)}(\rho)$ whose result is a reduced density operator representing the state of the environment. We give the definition of ground bisimulation and bisimilarity as follows.

Definition 3 ([9]). *A relation $\mathcal{R} \subseteq Con \times Con$ is a ground simulation if for any $\mathcal{C} = \langle P, \rho \rangle$, $\mathcal{D} = \langle Q, \sigma \rangle$, $\mathcal{C} \mathcal{R} \mathcal{D}$ implies that $qv(P) = qv(Q)$, $tr_{qv(P)}(\rho) = tr_{qv(Q)}(\sigma)$, and*

- *whenever $\mathcal{C} \xrightarrow{\alpha} \Delta$, there is some distribution Θ with $\mathcal{D} \xrightarrow{\hat{\alpha}} \Theta$ and $\Delta \mathcal{R}^\circ \Theta$.*

A relation \mathcal{R} is a ground bisimulation if both \mathcal{R} and \mathcal{R}^{-1} are ground simulations. We denote by \approx the largest ground bisimulation, called ground bisimilarity. If the above weak transition $\mathcal{D} \xrightarrow{\hat{\alpha}} \Theta$ is replaced by a strong transition $\mathcal{D} \xrightarrow{\alpha} \Theta$, we obtain a strong ground bisimulation.

In the rest of the paper, we mainly focus on ground bisimulation and only briefly mention the algorithm for checking strong ground bisimulation.

3 Algorithm

We present an on-the-fly algorithm to check if two configurations are ground bisimilar.

The algorithm maintains two sets *NonBisim* and *Bisim* to keep non-bisimilar and bisimilar state pairs, respectively. When the algorithm terminates, *Bisim* should contain all the state pairs satisfying the bisimulation relation.

The function **Bisim**(t, u), as shown in Algorithm 1, is the main function of the algorithm, which attempts to find the smallest bisimulation containing the pair (t, u). It initialises *Bisim* and a set named *Visited* to store the visited pairs, then calls the function **Match** to search for a bisimulation. The function **Match**($t, u, Visited$) invokes a depth-first traversal to match a pair of states (t, u) with all their possible behaviours. The set *Visited* is updated before the traversal for detecting loops. We also match the behaviours of t and u from both directions as we are checking bisimulations. Two states are deemed non-bisimilar in three cases:

- one state has a transition that cannot be matched by any possible weak transition from the other;
- they do not have the same set of free quantum variables;
- the density operators of them corresponding to their quantum registers are different.

The first case is checked by **MatchAction**, and the other two are done in **Match**. We add a pair of states to *NonBisim* if one of the three cases above has occurred. Otherwise, it will be stored in *Bisim*.

An auxiliary function **Act**(t) is invoked in **Match** to discover the next action that t can perform. If t have no more action to perform the function will return an empty set.

The function **MatchAction**($\alpha, t, u, Visited$) checks the equivalence of configurations through comparing their transitions. The function recursively discovers the next equivalent state pairs between the target states of the transitions. Technically, it checks the condition that if $t \xrightarrow{\alpha} \Delta$ then there exists some Θ such that $u \xrightarrow{\hat{\alpha}} \Theta$ and $\Delta \mathcal{R}^\circ \Theta$. Here we use as a subroutine a procedure of [28] to reduce the problem to a linear programming problem that can be solved in polynomial time. The problem is defined in Appendix. In **MatchAction**, we introduce a predicate **LP**($\Delta, u, \alpha, \mathcal{R}$) which is true if and only if the linear programming problem has a solution. We invoke the function **Close** to construct an equivalence relation \mathcal{R} between S and the states in the support of the target distribution. Note that in Lines 28 and 34 we have two distinct cases because in output actions the emitted values are required to be equal, which are unlike other types of actions.

In general, there are loops in pLTSs. When a state pair to be considered is already contained in *Visited* it will be assumed to be bisimilar and added to *Assumed* (Lines 42-43). Later on, if the pair of states are found to be non-bisimilar, the pair will be added to *NonBisim* and a wrong assumption exception (Lines 18-21) will be raised to restart the checking process from the original pair of states. Then **Bisim**(t, u) renews the sets *Bisim*, *Visited* and *Assumed* to remove the pairs checked under the wrong assumption (Lines 4-6).

Algorithm 1 Checking ground bisimulation

Require: Two pLTSs with initial configurations t and u .

Ensure: A boolean value b_{res} indicating if the two pLTSs are ground bisimilar.

```

1: function GroundBisimulation( $t, u$ ) =
2:   NonBisim :=  $\emptyset$ 
3:   function Bisim( $t, u$ ) = try {
4:     Bisim :=  $\emptyset$ 
5:     Visited :=  $\emptyset$ 
6:     Assumed :=  $\emptyset$ 
7:     return Match( $t, u, Visited$ )
8:   } catch WrongAssumptionException  $\Rightarrow$  Bisim( $t, u$ )
9:
10: function Match( $t, u, Visited$ )  $\triangleright t = \langle P, \rho \rangle$  and  $u = \langle Q, \sigma \rangle$ 

```

```

11:   Visited := Visited  $\cup$   $\{(t, u)\}$ 
12:   b :=  $\bigwedge_{\alpha \in \mathbf{Act}(t)}$  MatchAction( $\alpha, t, u, \text{Visited}$ )
13:    $\bar{b}$  :=  $\bigwedge_{\alpha \in \mathbf{Act}(u)}$  MatchAction( $\alpha, u, t, \text{Visited}$ )
14:    $b_{c_1}$  :=  $qv(P) = qv(Q)$ 
15:    $b_{c_2}$  :=  $tr_{qv(P)}(\rho) = tr_{qv(P)}(\sigma)$ 
16:    $b_{res}$  :=  $b \wedge \bar{b} \wedge b_{c_1} \wedge b_{c_2}$ 
17:   if  $b_{res}$  is tt then  $Bisim = Bisim \cup \{(t, u)\}$ 
18:   else if  $b_{res}$  is ff then
19:      $NonBisim = NonBisim \cup \{(t, u)\}$ 
20:     if  $(t, u) \in Assumed$  then
21:       raise WrongAssumptionException
22:   return  $b_{res}$ 
23:
24: function MatchAction( $\alpha, t, u, \text{Visited}$ )
25:   switch  $\alpha$  do
26:     case  $c!$ 
27:       for  $t \xrightarrow{c!e_i} \Delta_i$  do
28:         Assume  $\{t_k\}_{t_k \in [\Delta_i]}$  and  $\{u_j\}$ 
            $u \xrightarrow{c!e'_j} \Gamma \wedge e_i = e'_j \wedge u_j \in [\Gamma]$ 
29:          $\mathcal{R} := \{(t_k, u_j) \mid \mathbf{Close}(t_k, u_j, \text{Visited}) = \mathbf{tt}\}$ 
30:          $\theta := \mathbf{LP}(\Delta_i, u, \alpha, \mathcal{R})$ 
31:       return  $\bigwedge_i \theta_i$ 
32:     otherwise
33:       for  $t \xrightarrow{\alpha} \Delta_i$  do
34:         Assume  $\{t_k\}_{t_k \in [\Delta_i]}$  and  $\{u_j\}_{u_j \in \Gamma \wedge u_j \in [\Gamma]}$ 
35:          $\mathcal{R} := \{(t_k, u_j) \mid \mathbf{Close}(t_k, u_j, \text{Visited}) = \mathbf{tt}\}$ 
36:          $\theta := \mathbf{LP}(\Delta_i, u, \alpha, \mathcal{R})$ 
37:       return  $\bigwedge_i \theta_i$ 
38:
39: function Close( $t, u, \text{Visited}$ )
40:   if  $(t, u) \in Bisim$  then return tt
41:   else if  $(t, u) \in NonBisim$  then return ff
42:   else if  $(t, u) \in Visited$  then
43:      $Assumed = Assumed \cup \{(t, u)\}$ 
44:   return tt
45:   else return Match( $t, u, \text{Visited}$ )
    
```

Now let us prove the termination and correctness of the algorithm.

Theorem 1 (Termination). *Given two configurations t and u , the function **GroundBisimulation**(t, u) always terminates.*

Proof. The algorithm starts with two empty sets $NonBisim$ and $Bisim$. The next action to perform is detected in **Match**. Then it invokes function **MatchAction** to find the next new pair of configurations and recursively call function

Match to check them. Once a state pair is checked to be non-bisimilar in function **Match**, it is added into *NonBisim*. Meanwhile, if it is also contained in the set *Assumed*, the algorithm restarts a new execution of **Bisim**. Let k denote the number of executions of **Bisim**, and $NonBisim_k$ be the set *NonBisim* at the end of **Bisim** _{k} . It is easy to show by induction that $NonBisim_k \subset NonBisim_{k+1}$ for any $k \geq 0$. Since the system under consideration is finite-state, there always exists some n such that $NonBisim_n$ is the largest set of non-bisimilar state pairs and **Bisim** _{n} is the last execution of **Bisim**.

After the execution of **Bisim** _{n} , no more exceptions will be raised. Each time **Match** is executed with t and u as its parameters, we add (t, u) into *Visited*. The quantum variables and the configurations of the quantum registers for t and u are compared. When no more state pairs are added into *Visited*, the function *Match* will not be invoked again and the whole algorithm will terminate. \square

Theorem 2 (Correctness). *Given two configurations t and u from two pLTSs, **Bisim**(t, u) returns true if and only if they are ground bisimilar.*

Proof. Let **Bisim** _{n} be the last execution of **Bisim**. Let $NonBisim_n$ and $Bisim_n$ be the values of the two sets *NonBisim* and *Bisim*, respectively, recording the checked state pairs at the end of **Bisim** _{n} . By inspecting **Match**, we know that $NonBisim_n \cap Bisim_n = \emptyset$.

Let us analyse the result returned by **Bisim** _{n} , which is the output of the function call **Match**($t, u, Visited$). If the result is *false* then one of the conjuncts in b_{res} is invalid, which means that one of the three cases discussed in the beginning of Section 3 occurs, thus t and u are indeed non-bisimilar. If the return is *true* then there is $Bisim_n = Visited_n \setminus NonBisim_n$. For each pair $(t, u) \in Bisim_n$, all the conjuncts in b_{res} must be true. Both t and u must have the same set of free quantum variables and the same density operators. In addition, they have matching transitions. That is, for any action α , if $t \xrightarrow{\alpha} \Delta$ then there exists some weak distribution Θ such that $u \xRightarrow{\alpha} \Theta$ and $\Delta \mathcal{R}^\circ \Theta$. This is true because (i) the relation \mathcal{R} in function **MatchAction** is correctly constructed, and (ii) the lifted relation \mathcal{R}° exists. Below we argue for (i); the existence of the lifting operation in (ii) relies on the validity of the predicate **LP** whose correctness is established by Theorem 9 in [28].

The algorithm adds a pair into *Assumed* _{n} if the pair to be checked has already been visited and passed the bisimulation checking conditions. It implies that $Assumed_n \subseteq Visited_n$. Furthermore, as there is no wrong assumption detected after the execution of **Bisim** _{n} , we have $Assumed_n \subseteq Bisim_n$ which implies that $Bisim_n = Assumed_n \cup Bisim_n$. So $Bisim_n$ constitutes a bisimulation relation containing the initial state pair (t, u) . \square

Before concluding this section, we analyse the time complexity of the algorithm.

Theorem 3 (Complexity). *Let the number of configurations reachable from t and u be n . The time complexity of function **Bisim**(t, u) is polynomial in n .*

Proof. The number of state pairs is at most n^2 . The number of state pairs examined in the k th execution of **Bisim** is at most $O(n^2 - k)$. Therefore, the total number of state pairs examined is at most $O(n^2 + (n^2 - 1) + \dots + 1) = O(n^4)$. Note that each state has finitely many outgoing transitions. Given a transition, to check if there exists a weak matching transition, we call the function **LP** at most once, the construction of a flow network and solving the linear programming problem are both polynomial in n if we use the algorithm in [28]. Consequently, the whole algorithm is also polynomial in n . \square

For the strong version of ground bisimulation, we are only concerned with the matching of strong transitions. Therefore, Algorithm 1 can be simplified and there is no need of the predicate **LP** in the function **MatchAction**.

4 Implementation and Experiments

In this section, we report on an implementation of our approach and provide the experimental results of verifying several quantum communication protocols.

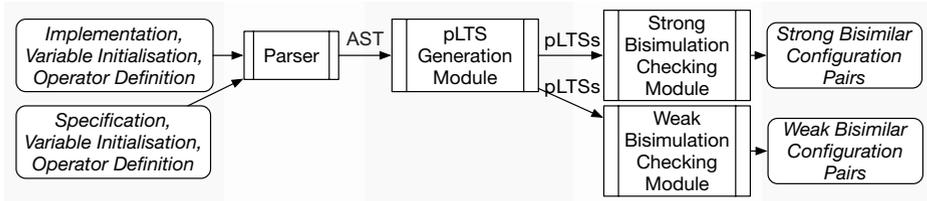


Fig. 3. Verification workflow.

4.1 Implementation

We have implemented both strong and weak ground bisimulation checkers in Python 3.7. The workflow of our tool is sketched in Figure 3. The tool consists of a pLTS generation module and two bisimulation checking modules, devoted to modeling and verification, respectively. The input of this tool is a specification and an implementation of a quantum protocol, both described as qCCS processes, the definition of user-defined operators, as well as an initialisation of classical and quantum variables. Unlike classical variables, the initialisation of all quantum variables, deemed as a quantum register, is accomplished at the same time so to allow for superposition states. The final output of the tool is a result indicating whether the specification and the implementation are bisimilar under the same initial states. The algorithm also stores the bisimilar state pairs and non-bisimilar state pairs in two tables.

The pLTS generation module acts as a preprocessing unit before the verification task. It first translates the input qCCS processes into two abstract syntax

trees (ASTs) by a parser. Then the ASTs are transformed into two pLTSs according to the operational semantics given in Figure 2, using the user-defined operators and the initial values of variables. The weak bisimulation checking module implements the weak ground bisimilarity checking algorithm we defined in the last section. It checks whether the initial states of the two generated pLTSs are weakly bisimilar.

The tool is available in [24], where we also provide all the examples for the experiments to be discussed in Section 4.3.

4.2 BB84 Quantum Key Distribution Protocol

To illustrate the use of our tool, we formalise the BB84 quantum key distribution protocol. Our formalisation follows [11], where a manual analysis of the protocol is provided. Now we perform automatic verification via the ground bisimulation checker.

The BB84 protocol provides a provably secure way to create a private key between two partners with a classical authenticated channel and a quantum insecure channel between them. The protocol does not make use of entangled states. It ensures its security through the basic property of quantum mechanics: if the states to be distinguished are not orthogonal, such as $|0\rangle$ and $|+\rangle$, then information gain about a quantum state is only possible at the expense of changing the state. Let the sender and the receiver be *Alice* and *Bob*, respectively. The basic BB84 protocol with a sequence of qubits \tilde{q} with size n goes as follows:

1. *Alice* randomly generates two sequences of bits \tilde{B}_a and \tilde{K}_a using her qubits \tilde{q} . Note that \tilde{q} here are auxiliary qubits which are not modified in this step.
2. *Alice* sets the state of \tilde{q} , such that the i th bits of \tilde{q} is $|x_y\rangle$ where x and y are the i th bits of \tilde{B}_a and \tilde{K}_a , and respectively, $|0_0\rangle = |0\rangle$, $|0_1\rangle = |1\rangle$, $|1_0\rangle = |+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ and $|1_1\rangle = |-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.
3. *Alice* sends her qubits \tilde{q} to *Bob*.
4. *Bob* randomly generates a sequence of bits \tilde{B}_b using his qubits \tilde{q}' .
5. *Bob* measures the i th qubit of \tilde{q} he received from *Alice* according to the basis determined by the i th bit of \tilde{B}_b . Respectively, the basis is $\{|0\rangle, |1\rangle\}$ if it is 0 and $\{|+\rangle, |-\rangle\}$ if it is 1.
6. *Bob* sends his choice of measurements \tilde{B}_b to *Alice*, and after receiving the information, *Alice* sends her \tilde{B}_a to *Bob*.
7. *Alice* and *Bob* match two sequences of bits \tilde{B}_a and \tilde{B}_b to determine at which positions the bits are equal. If the bits match, they keep the corresponding bits of \tilde{K}_a and \tilde{K}_b . Otherwise, they discard them.

After the execution of the basic BB84 protocol, the remaining bits of \tilde{K}_a and \tilde{K}_b should be the same, provided that the communication channels are perfect and there is no eavesdropper.

Implementation. For simplicity, we assume that the sequence \tilde{q} consists of only one qubit. This is enough to reflect the essence of the protocol. The other qubits

used below are auxiliary qubits for the operation *Ran*.

$$\begin{aligned}
 Alice &\stackrel{def}{=} Ran[q_1; B_a].Ran[q_1; K_a].Set_{K_a}[q_1].H_{B_a}[q_1].\underline{A2B}!q_1. \\
 &\quad b2a?B_b.a2b!B_a.key_a!cmp(K_a, B_a, B_b).\mathbf{nil}; \\
 Bob &\stackrel{def}{=} \underline{A2B}?q_1.Ran[q_2; B_b].M_{B_b}[q_1; K_b].b2a!B_b. \\
 &\quad a2b?B_a.key_b!cmp(K_b, B_a, B_b).\mathbf{nil}; \\
 BB84 &\stackrel{def}{=} (Alice||Bob) \setminus \{a2b, b2a, \underline{A2B}\}
 \end{aligned}$$

where there are several special operations:

- $Ran[q; x] = Set_+[q].M_{0,1}[q; x].Set_0[q]$, where Set_+ (resp. Set_0) is the operation which sets a qubit it applies on to $|+\rangle$ (resp. $|0\rangle$), $M_{0,1}[q; x]$ is the quantum measurement on q according to the basis $\{|0\rangle, |1\rangle\}$ and stores the result into x .
- $Set_K[q]$ sets the qubit q to the state $|K\rangle$.
- $H_B[q]$ applies H or does nothing on the qubit q depending on whether the value of B is 1 or 0.
- $M_B[q; K]$ is the quantum measurement on q according to the basis $\{|+\rangle, |-\rangle\}$ or $\{|0\rangle, |1\rangle\}$ depending on whether the value of B is 1 or 0.
- $cmp(x, y, z)$ returns x if y and z match, and ϵ , meaning it is empty, if they do not match.

Specification. The specification can be defined as follows using the same operations:

$$\begin{aligned}
 BB84_{spec} &\stackrel{def}{=} Ran[q_1; B_a].Ran[q_1; K_a].Ran[q_2; B_b] \\
 &\quad .(key_a!cmp(K_a, B_a, B_b).\mathbf{nil}||key_b!cmp(K_a, B_a, B_b).\mathbf{nil}).
 \end{aligned}$$

Input. For the implementation of *BB84*, we need to declare the following variables and operators in the input attached to it.

- The classical bits are named B_a, K_a for *Alice* and B_b, K_b for *Bob*.
- The qubits are declared together as a vector $|q_1, q_2\rangle$. The vector always needs an initial value. We can set it to be $|00\rangle$ in this example.

When modelling the protocol, we use several operators. They should be defined and their definitions are part of the input.

- The operator *Ran* involves two operators Set_+, Set_0 and a measurement $M_{0,1}$ measuring the qubit according to the basis $\{|0\rangle, |1\rangle\}$.
- Set_K needs Set_0 and Set_1 .
- H_B requires the Hadamard gate H .
- M_B uses the measurement $M_{+,-}$ which measures the qubit according to the basis $\{|+\rangle, |-\rangle\}$.

The function *cmp* is treated as an in-built function, so there is no need to define it in the input.

For the specification *BB84_{spec}*, we only declare the classical bits B_a, B_b, K_a , qubits q_1, q_2 and the operator *Ran*. The variables and operators declared here are the same as those in the input of the implementation.

Output. Taking the input discussed above, the tool first generates two pLTSs, with over 150 states for the implementation and 80 states for the specification, and then runs the ground bisimulation checking algorithm. As we can see from the fifth row in Table 1, our tool confirms that $\langle BB84, \rho_0 \rangle \approx \langle BB84_{spec}, \rho_0 \rangle$, where ρ_0 denotes the initial state of the quantum register, thus the implementation is faithful to the specification. In the output of the tool, there is an enumeration of 1084 pairs of non-bisimilar states and 3216 pairs of bisimilar states. The pLTSs and the state pairs can be found in [24].

4.3 Experimental Results

We conducted experiments on several quantum communication protocols with a few different input variables. Table 1 provides a summary of our experimental results obtained on a macOS machine with an Intel Core i7 2.5 GHz processor and 16GB of RAM.

Weak ground bisimulation							
Program	Variables	Bisi	Impl	Spec	N	B	ms
Super-dense coding	$q_1q_2 = 00\rangle, x = 1$	Yes	16	5	9	20	259
	$q_1q_2 = 00\rangle, x = 5$	No	6	2	-	-	2
Super-dense coding (modified)	$q_1q_2 = 00\rangle, x = 5$	Yes	8	5	5	12	110
Teleportation	$q_1q_2q_3 = 100\rangle$	Yes	34	3	22	22	232
	$q_1q_2q_3 = \frac{1}{\sqrt{2}} 000\rangle + \frac{1}{\sqrt{2}} 100\rangle$	Yes	34	3	22	22	264
	$q_1q_2q_3 = \frac{\sqrt{3}}{2} 000\rangle + \frac{1}{2} 100\rangle$	Yes	34	3	22	22	239
Secret Sharing	$q_1q_2q_3q_4 = 1000\rangle$	Yes	103	3	65	65	1339
	$q_1q_2q_3q_4 = \frac{1}{\sqrt{2}} 0000\rangle + \frac{1}{\sqrt{2}} 1000\rangle$	Yes	103	3	65	65	1252
	$q_1q_2q_3q_4 = \frac{\sqrt{3}}{2} 0000\rangle + \frac{1}{2} 1000\rangle$	Yes	103	3	65	65	1187
BB84	$q_1q_2 = 00\rangle$	Yes	152	80	1084	3216	130163
BB84 (with eavesdropper)	$q_1q_2q_3 = 000\rangle$	Yes	1180	352	121072	75392	55728587
B92	$q_1q_2 = 00\rangle$	Yes	64	80	466	1284	34522
E91	$q_1q_2q_3q_4 = 0000\rangle$	Yes	124	80	964	2676	113840

Table 1. Experimental results. The columns headed by **Impl** and **Spec** show the numbers of nodes contained in the generated pLTSs of the implementations and specifications, respectively. Column **N** shows the sizes of the sets of non-bisimilar state pairs and Column **B** shows the sizes of the sets of bisimilar state pairs. Column **ms** shows the time cost of the verification in milliseconds.

In each case, we report the final outcome (whether an implementation is ground bisimilar to its specification), the number of nodes in two pLTSs, the numbers of non-bisimilar and bisimilar state pairs in *NonBisim* and *Bisim*, respectively, as well as the verification time of our ground bisimulation checking algorithm. The time cost excludes the part of pLTS generation which takes around one second in all the examples.

Besides the protocol discussed in Section 4.2, we also verify other ones that make use of entangled qubits such as the teleportation and the quantum secret sharing protocol. For quantum key distribution protocols, we conduct experiments on the BB84, the B92 and the E91.

Not all the cases in Table 1 give the size of the set *NonBisim* of non-bisimilar state pairs, as the bisimulation checking algorithm may immediately terminate once a negative verification result is obtained, i.e. the two initial states are not bisimilar.

Data Availability Statement

The datasets generated and/or analyzed during the current study are available in the figshare repository: <https://doi.org/10.6084/m9.figshare.11874942.v1>.

5 Conclusion and Future Work

We have presented an on-the-fly algorithm to check ground bisimulation for quantum processes in qCCS, and a simpler algorithm for strong ground bisimulation. Based on the algorithms, we have developed a tool to verify quantum communication protocols modelled as qCCS processes. We have carried out experiments on several non-trivial quantum communication protocols from superdense coding to key distribution and found the tool helpful.

As to future work, several interesting problems remain to be addressed. For example, a limitation of the current work is to compare quantum processes with predetermined states of quantum registers. Indeed, there are occasions where one would expect two processes to be equivalent for arbitrary initial states. It is infeasible to enumerate all those states. Then the symbolic bisimulations proposed in [11] will be useful. We are considering to implement the algorithm for symbolic ground bisimulation, and then tackle the more challenging symbolic open bisimulation, both proposed in that work. Another problem occurs in the experiment of Section 4.2. The example tested one qubit instead of a sequence of qubits because more qubits lead to a drastic growth of the running time, which shows a limitation of the current approach of explicitly representing state spaces.

Appendix

Algorithm 1 needs to check the condition that if $t \xrightarrow{\alpha} \Delta$ then there exists some Θ such that $u \xrightarrow{\hat{\alpha}} \Theta$ and $\Delta \mathcal{R}^\circ \Theta$. We use as a subroutine a procedure of [28] to reduce the problem to a network flow problem that can be solved in polynomial time.

Technically, we construct a network graph $G(\Delta, u, \alpha, \mathcal{R}) = (V, E)$ defined as follows. Let S be the set of reachable states, and \mathcal{R} be a binary relation on the states.

Let Δ and \blacktriangledown be two vertices that represent the source and the sink of the network, respectively. For each visible action α , the set of vertices V is given

below

$$V = \{\Delta, \blacktriangledown\} \cup S \cup S^{tr} \cup S_\alpha \cup S_\alpha^{tr} \cup S_\perp \cup S_{\mathcal{R}}$$

where

$$\begin{aligned} S^{tr} &= \{v^{tr} | tr = v \xrightarrow{\beta} \Gamma, \beta \in \{\alpha, \tau\}\}; \\ S_\alpha &= \{v_\alpha | v \in S\}; \\ S_\alpha^{tr} &= \{v_\alpha^{tr} | v^{tr} \in S^{tr}\}; \\ S_\perp &= \{v_\perp | v \in S\}; \\ S_{\mathcal{R}} &= \{v_{\mathcal{R}} | v \in S\}. \end{aligned}$$

and the set of edges E is

$$E = \{(\Delta, u)\} \cup L_1 \cup L_\alpha \cup L_2 \cup L_\perp^\alpha \cup L_{\mathcal{R}}$$

where

$$\begin{aligned} L_1 &= \{(v, v^{tr}), (v^{tr}, v') | tr = v \xrightarrow{\tau} \Gamma, v' \in [\Gamma]\}; \\ L_\alpha &= \{(v, v_\alpha^{tr}), (v_\alpha^{tr}, v'_\alpha) | tr = v \xrightarrow{\alpha} \Gamma, v'_\alpha \in [\Gamma]\}; \\ L_2 &= \{(v_\alpha, v_\alpha^{tr}), (v_\alpha^{tr}, v'_\alpha) | tr = v_\alpha \xrightarrow{\tau} \Gamma, v'_\alpha \in [\Gamma]\}; \\ L_\perp^\alpha &= \{(u_\alpha, u_\perp) | u \in S\}; \\ L_{\mathcal{R}} &= \{(s_\perp, s'_{\mathcal{R}}), (s'_{\mathcal{R}}, \blacktriangledown) | (s, s') \in \mathcal{R}\}. \end{aligned}$$

For the invisible action τ , the definition is similar: $V = \{\Delta, \blacktriangledown\} \cup S \cup S^{tr} \cup S_\perp \cup S_{\mathcal{R}}$ and $E = \{(\Delta, u)\} \cup L_1 \cup L_\perp \cup L_{\mathcal{R}}$ where $L_\perp = \{(s, s_\perp) | s \in S\}$.

If α is a visible action, we consider the following linear programming problem associated to $G(\Delta, u, \alpha, \mathcal{R})$:

$$\max \sum_{(s,v) \in E} -f_{s,v}$$

subject to

$$\begin{aligned} f_{s,v} &\geq 0 && \text{for each } (s,v) \in E \\ f_{\Delta,u} &= 1 \\ f_{v_{\mathcal{R}}, \blacktriangledown} &= \Delta(v) && \text{for each } v \in S \\ \sum_{(s,v) \in E} f_{s,v} - \sum_{(v,w) \in E} f_{v,w} &= 0 && \text{for each } v \in V \setminus \{\Delta, \blacktriangledown\} \\ f_{v^{tr}, v'} - \Gamma(v') \cdot f_{v, v^{tr}} &= 0 && \text{for each } tr = v \xrightarrow{\tau} \Gamma \text{ and } v' \in [\Gamma] \\ f_{v_\alpha^{tr}, v'_\alpha} - \Gamma(v') \cdot f_{v, v_\alpha^{tr}} &= 0 && \text{for each } tr = v \xrightarrow{\alpha} \Gamma \text{ and } v' \in [\Gamma] \\ f_{v_\alpha^{tr}, v'_\alpha} - \Gamma(v') \cdot f_{v_\alpha, v_\alpha^{tr}} &= 0 && \text{for each } tr = v \xrightarrow{\tau} \Gamma \text{ and } v' \in [\Gamma] \end{aligned}$$

Note that the fourth constraint is referred to as the flow-conservation constraints. The last three constraints link the source state and the result distribution.

For the invisible action τ , the linear programming problem associated to the network $G(\Delta, u, \tau, \mathcal{R})$ is the same as above except that the last two constraints are dropped.

We denote by $\mathbf{LP}(\Delta, u, \alpha, \mathcal{R})$ the predicate that is true if and only if the linear programming problem above has a solution.

References

1. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. *Physical Review A* **70**(052328) (2004)
2. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: *Proceedings of the 19th IEEE Symposium on Logic in Computer Science*. pp. 415–425. IEEE Computer Society (2004)
3. Ardeshir-Larijani, E., Gay, S.J., Nagarajan, R.: Automated equivalence checking of concurrent quantum systems. *ACM Transactions on Computational Logic* **19**(4), 28:1–28:32 (2018)
4. Baier, C., Engelen, B., Majster-Cederbaum, M.E.: Deciding bisimilarity and similarity for probabilistic processes. *Journal of Computer and System Sciences* **60**(1), 187–231 (2000)
5. Bennett, C.H., Brassard, G.: Quantum cryptography: Public-key distribution and coin tossing. In: *Proceedings of the IEEE International Conference on Computer, Systems and Signal Processing*. pp. 175–179 (1984)
6. Davidson, T.A.S.: *Formal Verification Techniques using Quantum Process Calculus*. Ph.D. thesis, University of Warwick (2011)
7. Deng, Y.: *Semantics of Probabilistic Processes: An Operational Approach*. Springer (2015)
8. Deng, Y., Du, W.: A local algorithm for checking probabilistic bisimilarity. In: *Proceedings of the 4th International Conference on Frontier of Computer Science and Technology*. pp. 401–407. IEEE Computer Society (2009)
9. Deng, Y., Feng, Y.: Open bisimulation for quantum processes. In: *Proceedings of the 7th IFIP International Conference on Theoretical Computer Science*. *Lecture Notes in Computer Science*, vol. 7604, pp. 119–133. Springer (2012)
10. Feng, Y., Duan, R., Ji, Z., Ying, M.: Probabilistic bisimulations for quantum processes. *Information and Computation* **205**(11), 1608–1639 (2007)
11. Feng, Y., Deng, Y., Ying, M.: Symbolic bisimulation for quantum processes. *ACM Transactions on Computational Logic* **15**(2), 1–32 (2014)
12. 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)
13. Feng, Y., Hahn, E.M., Turrini, A., Zhang, L.: QPMC: A model checker for quantum programs and protocols. In: *Proceedings of the 20th International Symposium on Formal Methods*. *Lecture Notes in Computer Science*, vol. 9109, pp. 265–272. Springer (2015)
14. Fernandez, J.C., Mounier, L.: Verifying bisimulations “on the fly”. In: *Proceedings of the 3rd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*. pp. 95–110. North-Holland (1990)
15. Gay, S.J., Nagarajan, R.: Communicating quantum processes. In: Palsberg, J., Abadi, M. (eds.) *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 145–157 (2005)
16. Gay, S.J., Nagarajan, R., Papanikolaou, N.: QMC: A model checker for quantum systems. In: *Proceedings of the 20th International Conference on Computer Aided Verification*. *Lecture Notes in Computer Science*, vol. 5123, pp. 543–547. Springer (2008)
17. Hennessy, M., Lin, H.: Symbolic bisimulations. *Theoretical Computer Science* **138**(2), 353–389 (1995)

18. Jorrand, P., Lalire, M.: Toward a quantum process algebra. In: Proceedings of the 1st Conference on Computing Frontiers. pp. 111–119. ACM (2004)
19. Kissinger, A.: Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing. Ph.D. thesis, University of Oxford (2011)
20. Kubota, T., Kakutani, Y., Kato, G., Kawano, Y., Sakurada, H.: Semi-automated verification of security proofs of quantum cryptographic protocols. *Journal of Symbolic Computation* **73**, 192–220 (2016)
21. Lalire, M.: Relations among quantum processes: Bisimilarity and congruence. *Mathematical Structures in Computer Science* **16(3)**, 407–428 (2006)
22. Liu, T., Li, Y., Wang, S., Ying, M., Zhan, N.: A theorem prover for quantum hoare logic and its applications. CoRR **abs/1601.03835** (2016)
23. Milner, R.: *Communication and Concurrency*. Prentice-Hall (1989)
24. Qin, X., Deng, Y., Du, W.: QBisim (2020), <https://github.com/MartianQXD/QBisim>
25. Sangiorgi, D.: A theory of bisimulation for the pi-calculus. *Acta Informatica* **33(1)**, 69–97 (1996)
26. Selinger, P.: Towards a quantum programming language. *Mathematical Structures in Computer Science* **14(4)**, 527–586 (2004)
27. Shor, P., Preskill, J.: Simple proof of security of the BB84 quantum key distribution protocol. *Physical Review Letters* **85(2)**, 441–444 (2000)
28. Turrini, A., Hermans, H.: Polynomial time decision algorithms for probabilistic automata. *Information and Computation* **244**, 134–171 (2015)
29. Ying, M., Feng, Y., Duan, R., Ji, Z.: An algebra of quantum processes. *ACM Transactions on Computational Logic* **10(3)**, 1–36 (2009)
30. Ying, M.: *Foundations of Quantum Programming*. Morgan Kaufmann (2016)

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

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





Deciding the bisimilarity of context-free session types

Bernardo Almeida , Andreia Mordido , and Vasco T. Vasconcelos 

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

Abstract. We present an algorithm to decide the equivalence of context-free session types, practical to the point of being incorporated in a compiler. We prove its soundness and completeness. We further evaluate its behaviour in practice. In the process, we introduce an algorithm to decide the bisimilarity of simple grammars.

Keywords: Types, Type equivalence, Bisimulation, Algorithm

1 Introduction

Session types enhance the expressivity of traditional types for programming languages by allowing the description of structured communication on heterogeneously typed channels [14,15,24]. Traditional session types are *regular* in the sense that the sequences of communication actions admitted by a type are in the union of a regular language (for finite executions) and an ω -regular language (for infinite executions). Introduced by Thiemann and Vasconcelos, context-free session types liberate traditional session types from the shackles of tail recursion, allowing, for example, the safe serialization of arbitrary recursive datatypes [26].

Session types are often used to discipline interactions in concurrent programs. When associated to (bidirectional, heterogeneous) channels, session types describe the permitted patterns of interaction. For example, a type of the form

$$\mathbf{rec} \ x. \ +\{\mathbf{Leaf} : \mathbf{Skip}, \mathbf{Node} : !\mathbf{Int}; x; x\}$$

may describe one end of a communication channel. A process holding such a channel end must first *select* between choices **Leaf** and **Node**. If **Leaf** is chosen, then type **Skip** forwards the interaction to the continuation, if any. If no continuation is present, then interaction is over. Otherwise, the process must send an integer (**!Int**) followed by two trees, as witnessed by the recursive calls occurring after the choice of **Node**. A concurrent process holding the other end of the channel interacts via a *dual* type:

$$\mathbf{rec} \ y. \ \&\{\mathbf{Leaf} : \mathbf{Skip}, \mathbf{Node} : ?\mathbf{Int}; y; y\}$$

In this case the process must be ready to offer both choices, **Leaf** and **Node**. For the latter option, the process must further receive an integer (**?Int**), followed by two trees.

Regular languages cannot capture such behaviour. The best one can do with regular session types (and without resorting to channel passing) is to use a

regular type that allows transmitting trees, as well as many other non tree-like structures. The correct behaviour of processes interacting on such a channel would need to be checked at runtime [2,26].

If the algorithmic aspects of type equivalence for regular session types are well known (Gay and Hole propose an algorithm to decide subtyping [9], from which type equivalence can be derived), the same does not apply to context-free session types. Thiemann and Vasconcelos [26] show that the equivalence of context-free session types is decidable, by reducing the problem to the verification of bisimulation for Basic Process Algebra (BPA) which, in turn, was proved decidable by Christensen, Hüttel, and Stirling [6]. Even if the equivalence problem for context-free session types is known to be decidable, no algorithm has been proposed. Padovani [20] introduces a language with context-free session types that avoids the problem of checking the equivalence of types by requiring annotations in the source code. Annotations result in the structural alignment between code and types. This alignment—enforced by an explicit resumption process operator that breaks sequential composition in types—sidesteps the problem central to this paper: that of checking type equivalence. Furthermore, there are some basic equivalences on types that the compiler is not able to identify [20].

After the breakthrough by Christensen, Hüttel, and Stirling—a result that provides no immediate practical algorithm—the problem of deciding the equivalence of BPA terms has been addressed by several researchers [4,6,8,18]. Most of these works provide no practical algorithm that can be readily used, except the one by Czerwinski and Lasota where a polynomial time algorithm is presented that decides the bisimilarity of normed context-free processes in $\mathcal{O}(n^5)$ [8]. However, context-free session types are not necessarily normed, which precludes resorting to this algorithm, or using the original result by Baeten, Bergstra, and Klop [3], as well as improvements by Hirshfeld, Jerrum, and Moller [12,13]. Moreover, the complexity estimates for deciding bisimilarity in BPA process are not promising. Kiefer provided an EXPTIME lower bound for BPA bisimilarity by proving this problem is EXPTIME-hard [19], whereas Jančar has provided a double exponential upper bound for this problem and proved that its complexity is $\mathcal{O}(2^{2^{\text{pol}(n)}})$ [17].

The decidability of deterministic pushdown automata (DPDA) has also been subject of much study [16,22,23]. Several techniques have been proposed to solve the problem, but no immediate practical algorithm was available until Henry and Sénizergues provide an algorithm for this problem [10]. Its poor performance however precludes its incorporation in a compiler. Furthermore, the algorithm Henry and Sénizergues propose handles the problem of language equivalence rather than the problem of deciding bisimilarity of DPDAs.

Our algorithm to decide the equivalence of context-free session types also allows deciding the bisimilarity of simple grammars (i.e., deterministic grammars in Greibach Normal Formal). It proceeds in three stages. The *first stage* builds a context-free grammar in Greibach Normal Formal (GNF)—in fact a simple grammar—from two context-free session types in a way that bisimulation is preserved. A basic result from Baeten, Bergstra, and Klop states that any

guarded BPA system can be transformed into Greibach Normal Form (GNF) while preserving bisimulation equivalence, but unfortunately no procedure is presented [3]. The *second stage* prunes the grammar by removing unreachable symbols in unnormed sequences of non-terminal symbols. This stage builds on the result of Christensen, Hüttel, and Stirling [6]. The *third stage* constructs an expansion tree, by alternating between expansion and simplification steps. This last stage uses expansion operations proposed by Jančar, Moller, and Hirschfeld [11,18], and simplification rules proposed by Caucal, Christensen, Hüttel, Stirling, Jančar, and Moller [5,6,18]. The finite representation of bisimulations of BPA transition graphs [5,6] is paramount for our results of soundness and completeness.

The branching nature of the expansion tree confers an (at least) exponential complexity to the algorithm. However, our experiments with a concrete implementation—both as a stand-alone tool and incorporated in a compiler [2]—are promising. We propose heuristics that decrease the execution time in 89% and reduce the number of timeouts by 95% (see Section 5).

We present an algorithm to decide the equivalence of context-free session types, practical to the point of being readily included in any compiler, an exercise that we conducted in parallel [2]. The main contributions of this work are:

- The proposal and implementation of an algorithm to decide type equivalence of context-free session types;
- A proof of its soundness and completeness against the declarative definition;
- The proposal and implementation of an algorithm to decide the bisimilarity of simple grammars; and
- The empirical study of the runtime behaviour of the implementation.

The rest of the paper is organized as follows: an introduction to context-free session types can be found in Section 2, the algorithm in Section 3, the main results in Section 4, evaluation in Section 5, and conclusions in Section 6.

2 Context-free session types

This section briefly introduces context-free session types, based on the work of Thiemann and Vasconcelos [26]. The types we consider build upon a denumerable set of *variables* and a set of *choice labels*. Metavariables X, Y, Z range over variables and ℓ over labels. We assume given a set of base types denoted by B . The syntax of types is given by the grammar below.

$$\begin{aligned} S, T ::= & \text{skip} \mid \sharp B \mid \star \{\ell_i : T_i\}_{i \in I} \mid S; T \mid \mu X. T \mid X \\ \sharp ::= & ! \mid ? \qquad \star ::= \oplus \mid \& \end{aligned}$$

In type $\mu X. T$, variable X is bound in the subterm T . The sets of bound and free variables in a given type are defined accordingly. Notation $[T/X]S$ denotes the resulting of substituting T for the (free) occurrences of X in S .

Judgement $S \checkmark$ characterizes *terminated* types: context-free session types that exhibit no further action [1].

Terminated predicate:

 $T\checkmark$

$$\text{skip}\checkmark \quad X\checkmark \quad \frac{S\checkmark \quad T\checkmark}{S;T\checkmark} \quad \frac{T\checkmark}{\mu X.T\checkmark}$$

Notice that all types of the form $\mu X.\mu X_1 \dots \mu X_n.X$, for $n \geq 0$, are terminated.

We are not interested in all types generated by the above grammar. If Δ is a list of pairwise distinct variables, then judgement $\Delta \vdash T$ characterises the types of interest: the *well-formed* types.

Type formation system:

 $\Delta \vdash T$

$$\frac{}{\Delta \vdash \text{skip}} \quad \frac{}{\Delta \vdash \sharp B} \quad \frac{X \in \Delta}{\Delta \vdash X} \quad \frac{\Delta \vdash S \quad \Delta \vdash T}{\Delta \vdash S;T} \quad \frac{\Delta \vdash T_i \quad (\forall i \in I)}{\Delta \vdash \star\{\ell_i : T_i\}_{i \in I}} \quad \frac{\neg T\checkmark \quad \Delta, X \vdash T}{\Delta \vdash \mu X.T}$$

Terminated processes have a simple characterisation—types comprising `skip`, μ and semicolon—which justifies the inclusion of $\neg T\checkmark$ in the rules for type formation (Thiemann and Vasconcelos [26] introduce a contractive judgement for the effect). Type formation serves two main purposes: ensuring that all variables introduced by μ -types are pairwise distinct and that types underneath a μ are not terminated. This can be clearly seen by formation rule for μ -types, where notation Δ, X is understood as requiring $X \notin \Delta$. In the sequel we assume that all types are such that $\vdash T$ and denote by \mathcal{T} the set such types.

The set of *actions* is generated by the following grammar.

$$a ::= \sharp B \mid \star \ell$$

The *labelled transition system* (LTS) for context-free session types is given by \mathcal{T} as the set of *states*, the set of *actions*, and the *transition relation* $S \xrightarrow{a}_{\mathcal{T}} T$ defined by the rules below.

Labelled transition system:

 $S \xrightarrow{a}_{\mathcal{T}} T$

$$\begin{array}{c} \sharp B \xrightarrow{\sharp B}_{\mathcal{T}} \text{skip} \quad \star\{\ell_i : T_i\}_{i \in I} \xrightarrow{\star \ell_j}_{\mathcal{T}} T_j \quad (j \in I) \\ \frac{S \xrightarrow{a}_{\mathcal{T}} S'}{S;T \xrightarrow{a}_{\mathcal{T}} S';T} \quad \frac{S\checkmark \quad T \xrightarrow{a}_{\mathcal{T}} T'}{S;T \xrightarrow{a}_{\mathcal{T}} T'} \quad \frac{[\mu X.S/X]S \xrightarrow{a}_{\mathcal{T}} T}{\mu X.S \xrightarrow{a}_{\mathcal{T}} T} \end{array}$$

Type bisimulation is defined in the usual way from the labelled transition system [21]. We say that a type relation \mathcal{R} is a *bisimulation* if, whenever $S\mathcal{R}T$, for all a we have:

- for each S' with $S \xrightarrow{a}_{\mathcal{T}} S'$, there is T' such that $T \xrightarrow{a}_{\mathcal{T}} T'$ and $S'\mathcal{R}T'$, and
- for each T' with $T \xrightarrow{a}_{\mathcal{T}} T'$, there is S' such that $S \xrightarrow{a}_{\mathcal{T}} S'$ and $S'\mathcal{R}T'$.

We say that two types are bisimilar, written $S \sim_{\mathcal{T}} T$, if there is a bisimulation \mathcal{R} with $S\mathcal{R}T$.

3 An algorithm to decide type bisimilarity

This section presents an algorithm to decide whether two types are in a type bisimulation. In the process we also provide an algorithm to decide the bisimilarity of simple context-free languages. The algorithm comprises three stages:

1. Translate the two types to a simple grammar,
2. Prune unreachable symbols, and
3. Explore an expansion tree, alternating between simplification and expansion operations, until finding an empty node—case in which it decides positively— or failing to expand all nodes—case in which it decides negatively.

Translating types to grammars. Type variables X are the *non-terminal symbols* and LTS labels a are the *terminal symbols*. Sequences of type variables \vec{X} are called *words*; ε denotes the empty word. A context-free grammar in Greibach Normal Form is a pair (\vec{X}, \mathcal{P}) where \vec{X} is the *start word* and \mathcal{P} a *set of productions* of the form $Y \rightarrow a\vec{Z}$ (context-free session types do not require productions of the form $Y \rightarrow \varepsilon$). Due to the deterministic nature of context-free session types, the grammars we are interested in are *simple*: for each non-terminal symbol Y and terminal symbol a , there is at most one production of the form $Y \rightarrow a\vec{Z}$.

Grammars in Greibach normal form naturally induce a labelled transition system by taking words \vec{X} for states, terminal symbols a for actions, and $\xrightarrow{a}_{\mathcal{P}}$, defined as $X\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Z}\vec{Y}$ when $X \rightarrow a\vec{Z} \in \mathcal{P}$, for the transition relation. The associated bisimilarity is denoted by $\sim_{\mathcal{P}}$.

The *unravelling* function on well-formed context-free session types, taken from Thiemann and Vasconcelos [26], is defined as follows.

$$\begin{aligned} \text{unr}(\mu X.T) &= \text{unr}([\mu X.T/X]T) \\ \text{unr}(S;T) &= \begin{cases} \text{unr}(T) & \text{unr}(S) = \text{skip} \\ (\text{unr}(S);T) & \text{unr}(S) \neq \text{skip} \end{cases} \\ \text{unr}(T) &= T \quad \text{in all other cases} \end{aligned}$$

The function terminates under the assumption that types are well formed.

Another function, *word*, builds a word from a type. In the process it updates a global set \mathcal{P} of grammar productions. Word concatenation is denoted by $\vec{X} \cdot \vec{Y}$.

$$\begin{aligned} \text{word}(\text{skip}) &= \varepsilon \\ \text{word}(S;T) &= \text{word}(S) \cdot \text{word}(T) \\ \text{word}(\sharp B) &= Y, \text{ setting } \mathcal{P} := \mathcal{P} \cup \{Y \rightarrow \sharp B\} \quad (Y \text{ fresh}) \\ \text{word}(\star\{\ell_i : T_i\}_{i \in I}) &= Y, \text{ setting } \mathcal{P} := \mathcal{P} \cup \{Y \rightarrow \star\ell_i \cdot \text{word}(T_i) \mid i \in I\} \quad (Y \text{ fresh}) \\ \text{word}(X) &= X \\ \text{word}(\mu X.T) &= X \end{aligned}$$

The following lemma relates terminated types to the result of a call to *word*.

Lemma 1. *Let $\vdash T$. Then, $T \checkmark$ if and only if $\text{word}(T) = \varepsilon$.*

Proof. The direct implication follows by rule induction on predicate \checkmark :

- Case $\text{skip}\checkmark$: $\text{word}(\text{skip}) = \varepsilon$.
- Case $X\checkmark$: if T is X , then $\not\checkmark T$.
- Case $S;T\checkmark$: by induction hypothesis on the rule premises $S\checkmark$ and $T\checkmark$, $\text{word}(S) = \varepsilon$ and $\text{word}(T) = \varepsilon$. Hence, $\text{word}(S;T) = \varepsilon$.
- Case $\mu X.S$: the hypothesis $T\checkmark$ and the rule premises of hypothesis $\vdash T$ are contradictory.

Conversely, if $\text{word}(T) = \varepsilon$, using the rules of the definition of word that produce the empty word:

- if T is skip , then we have $T\checkmark$.
- if T is $U;V$, $\text{word}(U) = \varepsilon$, and $\text{word}(V) = \varepsilon$, then, by induction, we have $U\checkmark$ and $V\checkmark$. Hence, $T\checkmark$.
- No other case in function word produces an empty word. \square

To define the translation of context-free session types to simple grammars, assume that $\{\mu X_1.T_1, \dots, \mu X_n.T_n\}$ is the set of all μ -subterms in a given type T . Further assume that $i < j$ whenever $X_j \in \text{free}(\mu X_i.T_i)$. That is, the μ -subterms are topologically sorted with respect to their lexical nesting, innermost subterms first. Now we identify unrolled versions of the μ -subterms.

$$\begin{aligned} T'_1 &= [\mu X_n.T_n/X_n] \cdots [\mu X_2.T_2/X_2][\mu X_1.T_1/X_1]T_1 \\ T'_2 &= [\mu X_n.T_n/X_n] \cdots [\mu X_2.T_1/X_2]T_2 \\ &\vdots \\ T'_n &= [\mu X_n.T_n/X_n]T_n \end{aligned}$$

Clearly each type T'_i is closed (has no free variables). Notice that if T is a μ -type, then $\mu X_n.T_n$ is T itself.

Finally, given an initial set of productions \mathcal{P}_0 , function grm translates a type T into a grammar composed of a start word and set of productions:

$$\text{grm}(T, \mathcal{P}_0) = (\text{word}(T), \mathcal{P}_n)$$

where each \mathcal{P}_i is computed from \mathcal{P}_{i-1} by the following recurrence,

$$\mathcal{P}'_i \cup \{X_i \rightarrow a_j \vec{Y}_j \vec{Z} \mid (Z \rightarrow a_j \vec{Y}_j) \in \mathcal{P}'_i\} \text{ where } (Z \vec{Z}, \mathcal{P}'_i) = \text{grm}(\text{unr}(T'_i), \mathcal{P}_{i-1})$$

Notice that $\text{word}(\text{unr}(T'_i))$ is a non-empty word because of Lemma 1 and the fact that each T'_i is non-terminated by hypothesis. The function grm terminates on all inputs (because recursion is always on subterms) and adds a finite number of productions to the original set. Furthermore, because choices in session types do not contain duplicated labels, the function returns a simple grammar.

To run grm on two well-formed types proceed as follows: rename the second type so that bound variables do not overlap with those of the first; start with an empty set of productions; run the algorithm consecutively on the two types to obtain two initial words and a single set of productions.

Example 1. Consider the following pair of context-free session types.

$$\begin{aligned} S &\triangleq (\mu X_1. \&\{n : X_1; X_1; ?\text{int}, \ell : ?\text{int}\}); (\mu X_2. !\text{int}; X_2; X_2) \\ T &\triangleq (\mu Y_1. \&\{n : Y_1; Y_1, \ell : \text{skip}\}; ?\text{int}); (\mu Y_2. !\text{int}; Y_2) \end{aligned}$$

Starting from the empty set of productions, running `grm` consecutively on S and on T produces the following set of productions

$$\begin{array}{llll} X_1 \rightarrow \&n X_1 X_1 X_3 & X_3 \rightarrow ?\text{int} & Y_1 \rightarrow \&n Y_1 Y_1 Y_3 & Y_2 \rightarrow !\text{int} Y_2 \\ X_1 \rightarrow \&\ell X_4 & X_4 \rightarrow ?\text{int} & Y_1 \rightarrow \&\ell Y_3 & Y_3 \rightarrow ?\text{int} \\ X_2 \rightarrow !\text{int} X_2 X_2 & & & & \end{array}$$

and two start words $X_1 X_2$ and $Y_1 Y_2$.

Pruning unnormed productions. For \vec{a} a non-empty sequence of non-terminal symbols a_1, \dots, a_n , write $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$ when $\vec{Y} \xrightarrow{a_1}_{\mathcal{P}} \dots \xrightarrow{a_n}_{\mathcal{P}} \vec{Z}$. We say that \vec{Y} is *normed* when $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$ for some \vec{a} , and that \vec{Y} is *unnormed* otherwise. When \vec{Y} is normed, the *minimal path* of \vec{Y} is the shortest \vec{a} such that $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$. In this case, the *norm* of \vec{Y} , denoted by $|\vec{Y}|$, is the length of \vec{a} . As observed by Christensen, Hüttel, and Stirling [6], any unnormed word \vec{Y} is bisimilar to its concatenation with any other word, that is, if \vec{Y} is unnormed, then $\vec{Y} \sim_{\mathcal{P}} \vec{Y} \vec{X}$. We use this fact to prune unreachable symbols in unnormed words. And we do this in all productions.

Example 2. Recall Example 1 and notice that X_2 and Y_2 are both unnormed. Then, the last occurrence of X_2 in production $X_2 \rightarrow !\text{int} X_2 X_2$ is unreachable, hence we simplify the production to obtain $X_2 \rightarrow !\text{int} X_2$.

Building an expansion tree. We base the third stage of the algorithm on the notion of *expansion tree* as proposed by Jančar and Moller [18], adapting an idea by Hirshfeld [11]. The *nodes* in trees are labelled by sets of pairs of words. We say that a node N' is an *expansion* of N if N' is a minimal set such that: for every pair $(\vec{X}, \vec{Y}) \in N$,

- if $\vec{X} \rightarrow a\vec{X}'$ then $\vec{Y} \rightarrow a\vec{Y}'$ with $(\vec{X}', \vec{Y}') \in N'$, and
- if $\vec{Y} \rightarrow a\vec{Y}'$ then $\vec{X} \rightarrow a\vec{X}'$ with $(\vec{X}', \vec{Y}') \in N'$.

An *expansion tree* is built from a root node: the singleton set containing the pair of start words obtained by translating the two types into a grammar. A children node is obtained from its parent node by expansion. However, as Jančar and Moller observed, expansions alone often lead to infinite trees. We then alternate between expansion and simplification operations, until either finding an empty node—case in which we decide equivalence positively—or failing to expand all nodes—case in which we decide equivalence negatively. We say that a branch is *successful* if it is infinite or finishes in an empty node, otherwise it is said to be *unsuccessful*.

In the *expansion step*, each node N derives a single child node, obtained as an expansion of N . As we are dealing with simple grammars, no branching is expected in the expansion tree at this step.

The *simplification step* consists on the application of the following rules:

Reflexive rule: Omit from a node any pair of the form (\vec{X}, \vec{X}) ;

Congruence rule: Omit from a node N any pair that belongs to the least congruence containing the ancestors of N ;

BPA1 rule: If $(X_0\vec{X}, Y_0\vec{Y})$ is in N and $(X_0\vec{X}', Y_0\vec{Y}')$ belongs to the ancestors of N , then create a sibling node for N replacing $(X_0\vec{X}, Y_0\vec{Y})$ by (\vec{X}, \vec{X}') and (\vec{Y}, \vec{Y}') ;

BPA2 rule: If $(X_0\vec{X}, Y_0\vec{Y})$ is in N and X_0 and Y_0 are normed, then:

Case $|X_0| \leq |Y_0|$: Let \vec{a} be a minimal path for X_0 and \vec{Z} the word such that $Y_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including the pairs $(X_0\vec{Z}, Y_0)$ and $(\vec{X}, \vec{Z}\vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$;

Otherwise: Let \vec{a} be a minimal path for Y_0 and \vec{Z} the word such that $X_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including the pairs $(X_0, Y_0\vec{Z})$ and $(\vec{Z}\vec{X}, \vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$.

Contrarily to expansion and to the reflexive and congruence simplifications, BPA rules promote branching in the expansion tree. We iteratively apply the simplification rules to ensure the algorithm computes the simplest possible children nodes derived from N . We can easily show that the simplification function that results from applying the reflexive, congruence, and BPA rules, has a fixed point in the complete partial ordered set of pairs node-ancestors, where the set of ancestors is fixed. The proof builds a partial order on the sets of pairs node-ancestors and uses Tarski's fixed point theorem [25]. The number of children nodes generated by the application of these rules is finite [6,18]. Notice that the sibling nodes do not exclude the (often) infinite branch resulting from successive expansions.

Checking the bisimilarity of simple grammars. Given a set of productions and two start words \vec{X} and \vec{Y} (all pruned), function `bisimG` alternates between simplification and expansion stages, starting with expansion. To avoid getting stuck in an infinite branch of the expansion tree, we use a breadth-first search on the expansion tree: node-ancestor pairs to be processed are stored in a queue. The initial pair inserted in the queue contains the initial node $\{(\vec{X}, \vec{Y})\}$ and an empty set of ancestors.

$$\text{bisimG}(\vec{X}, \vec{Y}, \mathcal{P}) = \text{expand}(\text{singletonQueue}(\{(\vec{X}, \vec{Y})\}, \emptyset), \mathcal{P})$$

Predicate `expand` terminates as soon as all nodes fail to expand (signalled by an empty queue), case in which the algorithm returns **False**, or an empty node is reached, case in which the algorithm returns **True**. Otherwise, it extracts node

n at the front of the queue, simplifies its child node, and recurs.

```

expand( $q, \mathcal{P}$ ) =
  if empty( $q$ ) then False
  else ( $n, a$ ) = front( $q$ )
    if empty( $n$ ) then True
    else if hasChild( $n, \mathcal{P}$ )
      then expand(simplify( $\{(child(n, \mathcal{P}), a \cup n)\}, dequeue(q), \mathcal{P}$ ))
      else expand(dequeue( $q$ ),  $\mathcal{P}$ )

```

The simplification stage distinguishes the case where all type variables are normed, in which case BPA1 is not required to decide equivalence [5,6], from the case where some type variables might be unnormed.

```

rules = if allProductionsNormed( $\mathcal{P}$ ) then [reflex, congruence, bpa2]
        else [reflex, congruence, bpa1, bpa2]

```

Function `simplify` applies the various rules iteratively, until reaching a fixed point. The application of the rules (via function `apply`) produces a set of nodes that are then enqueued. The simplification stage does not introduce new levels in the tree, hence the set of ancestors na is passed to function `apply` as is.

```

simplify( $na, q, \mathcal{P}$ ) = fold(enqueue,  $q$ , apply( $na$ , rules,  $\mathcal{P}$ ))

```

Example 3. The expansion tree for our running example is in Figure 1. Once a successful branch is reached (marked with \checkmark), `bisimG($\vec{X}, \vec{Y}, \mathcal{P}$)` returns **True**.

Checking the bisimilarity of context-free session types. Function `bisimT` decides the equivalence of two well-formed and renamed types, S and T . It starts by computing the start words for S and T by first translating S to a grammar and enriching this with the productions for type T . After pruning the productions in the grammar (function `prune`), the equivalence of S and T is decided using function `bisimG`.

```

bisimT( $T, U$ ) = bisimG( $\vec{X}, \vec{Y}, \text{prune}(\mathcal{P})$ )
                where ( $\vec{X}, \mathcal{P}'$ ) = grm( $S, \emptyset$ )
                      ( $\vec{Y}, \mathcal{P}$ ) = grm( $T, \mathcal{P}'$ )

```

4 Correctness of the algorithm

In this section we prove that function `bisimT` is sound and complete with respect to the meta-theory of context-free session types. We start by showing a full abstraction result between context-free session types and grammars in Greibach Normal Form. Then, based on results from Caucal [5], Christensen, Hüttel, and Stirling [6], Jančar and Moller [18], we conclude that the algorithm we propose is sound and complete.

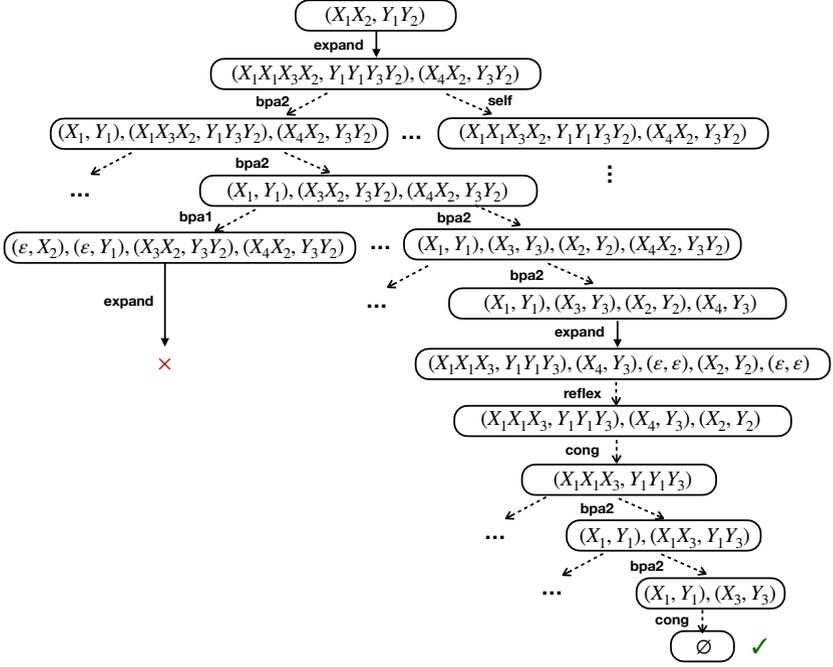


Fig. 1: An example of an expansion tree

Type translation is fully abstract. Sections 2 and 3 introduce bisimulation relations on the set \mathcal{T} of types $\sim_{\mathcal{T}}$ and on a given set \mathcal{P} of productions $\sim_{\mathcal{P}}$. Our ultimate goal is to prove that we can faithfully analyze the bisimilarity of types by analyzing the bisimilarity of the corresponding grammars. For this purpose, we prove that the translation proposed in Section 3 is a *fully abstract encoding*, i.e., preserves the bisimilarity relation.

We start showing that the transformation of types to grammars preserves the labelled transitions. The following result states that grammars produced by `grm` mimic the transitions of the corresponding types and vice-versa.

Lemma 2. *Let $(\vec{X}, \mathcal{P}') = \text{grm}(S, \emptyset)$ and $(\vec{Y}, \mathcal{P}) = \text{grm}(T, \mathcal{P}')$. Then, $S \xrightarrow{a}_{\mathcal{T}} T$ if and only if $\vec{X} \xrightarrow{a}_{\mathcal{P}} \vec{Y}$.*

Proof. For the direct implication we proceed by rule induction on the hypothesis, using the definition of word.

- Case $\#B$: if $\#B \xrightarrow{\#B}_{\mathcal{T}} \text{skip}$, then $\text{word}(\#B) \xrightarrow{\#B}_{\mathcal{P}} \varepsilon$.
- Case $\star\ell_i$: if $\star\{\ell_i : S_i\}_{i \in I} \xrightarrow{\star\ell_i}_{\mathcal{T}} S_i$, then $\text{word}(S) \xrightarrow{\star\ell_i}_{\mathcal{P}} \text{word}(S_i)$.
- Case $S_1; S_2$ with $S_1 \xrightarrow{a}_{\mathcal{T}} S'_1$: if $S_1; S_2 \xrightarrow{a}_{\mathcal{T}} S'_1; S_2$ and $S_1 \xrightarrow{a}_{\mathcal{T}} S'_1$, by induction hypothesis, we have $\text{word}(S_1) \xrightarrow{a}_{\mathcal{P}} \text{word}(S'_1)$. Furthermore, $\text{word}(S_1; S_2) = \text{word}(S_1) \cdot \text{word}(S_2)$. Hence, $\text{word}(S_1; S_2) \xrightarrow{a}_{\mathcal{P}} \text{word}(S'_1; S_2)$.

- Case $S_1; S_2$ with $S_1 \checkmark$ and $S_2 \xrightarrow{a}_{\mathcal{T}} S'_2$: in the case $S_1; S_2 \xrightarrow{a}_{\mathcal{T}} S'_2$, where $S_1 \checkmark$ and $S_2 \xrightarrow{a}_{\mathcal{T}} S'_2$, by Lemma 1 and since $S_1 \checkmark$, we have $\text{word}(S_1; S_2) = \text{word}(S_2)$. Thus, by induction hypothesis we have $\text{word}(S_1; S_2) \xrightarrow{a}_{\mathcal{P}} \text{word}(S'_2)$.
- Case $\mu X.T$: if $\mu X.T \xrightarrow{a}_{\mathcal{T}} S'$, then $[\mu X.T/X]T \xrightarrow{a}_{\mathcal{T}} S'$. Also $\text{unr}(S) \xrightarrow{a}_{\mathcal{T}} S'$ and, by induction hypothesis, $\text{word}(\text{unr}(S)) \xrightarrow{a}_{\mathcal{P}} \text{word}(S')$. Hence, by definition of word , $\text{word}(S) = X \xrightarrow{a}_{\mathcal{P}} \text{word}(S')$.

For the reverse implication, we prove that any transition in the grammar leads to a transition in the corresponding types.

- if $\text{word}(S) \xrightarrow{\sharp B}_{\mathcal{P}} \vec{X}$, then $\text{word}(S) = Y \cdot \vec{X}$, where $Y \xrightarrow{\sharp B}_{\mathcal{P}} \varepsilon$, and so $\text{unr}(S) = \sharp B; T$ and thus $S \xrightarrow{\sharp B}_{\mathcal{T}} T$.
- if $\text{word}(S) \xrightarrow{* \ell_i}_{\mathcal{P}} \vec{X}$, then $\text{word}(\text{unr}(S)) = Y$, where $Y \xrightarrow{* \ell_i}_{\mathcal{P}} \vec{X}$. Hence, $\text{unr}(S)$ is of the form $\star \{\ell'_j : U_j\}_{j \in J}; T$ with $\ell_i = \ell'_j$ and $\vec{X} = \text{word}(U_j; T)$, for some $j \in J$. Using the LTS we conclude that $S \xrightarrow{* \ell_i}_{\mathcal{T}} U_j; T$. \square

Lemma 3. *If $\text{word } S \xrightarrow{a}_{\mathcal{P}} \vec{X}$, then exists T s.t. $S \xrightarrow{a}_{\mathcal{T}} T$ and $\vec{X} = \text{word } T$.*

Proof. By induction on the definition of word . \square

The main result of this subsection follows from Lemmas 2 and 3.

Theorem 1. *Let $(\vec{X}, \mathcal{P}') = \text{grm}(S, \emptyset)$ and $(\vec{Y}, \mathcal{P}) = \text{grm}(T, \mathcal{P}')$. Then, grm is a full abstract encoding, i.e., $S \sim_{\mathcal{T}} T$ if and only if $\vec{X} \sim_{\mathcal{P}} \vec{Y}$.*

Proof. For the direct implication, assume that $S \sim_{\mathcal{T}} T$ and let \mathcal{B} be a bisimulation for S and T . Then, consider $\mathcal{B}' = \{(\text{word}(S_0), \text{word}(T_0)) \mid (S_0, T_0) \in \mathcal{B}\}$. Obviously, $(\text{word}(S), \text{word}(T)) \in \mathcal{B}'$. To prove that \mathcal{B}' is a bisimulation, one assumes that $\text{word}(S_0) \xrightarrow{a}_{\mathcal{P}} \vec{X}$ and proves that there exists \vec{Y} such that $\text{word}(T_0) \xrightarrow{a}_{\mathcal{P}} \vec{Y}$ with $(\vec{X}, \vec{Y}) \in \mathcal{B}'$. This proof is done by coinduction on the definition of word , uses Lemmas 2, 3, and the definition of \mathcal{B}' .

For the reverse implication, assume that $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, with $\vec{X} = \text{word}(S)$ and $\vec{Y} = \text{word}(T)$ and let \mathcal{B}' be a bisimulation for \vec{X} and \vec{Y} . Then, consider $\mathcal{B} = \{(S_0, T_0) \mid (\text{word}(S_0), \text{word}(T_0)) \in \mathcal{B}'\}$. Notice that $(S, T) \in \mathcal{B}$. The proof that \mathcal{B} is a bisimulation, consists in showing that: given $(S_0, T_0) \in \mathcal{B}$, such that $S_0 \xrightarrow{a}_{\mathcal{T}} S'_0$, there exists T'_0 such that $T_0 \xrightarrow{a}_{\mathcal{T}} T'_0$ and $(S'_0, T'_0) \in \mathcal{B}$. The proof follows by rule coinduction on the LTS and uses Lemmas 2 and 3. \square

Now we sketch the proof that pruning grammars also preserves bisimulation. We distinguish the grammars in the context through the subscript of \sim .

Theorem 2. *$\vec{X} \sim_{\mathcal{P}} \vec{Y}$ if and only if $\vec{X} \sim_{\text{prune}(\mathcal{P})} \vec{Y}$.*

Proof. For the direct implication, the bisimulation for \vec{X} and \vec{Y} over \mathcal{P} is also a bisimulation for \vec{X} and \vec{Y} over $\text{prune}(\mathcal{P})$. For the reverse implication, if \mathcal{B}' is a bisimulation for \vec{X} and \vec{Y} over $\text{prune}(\mathcal{P})$, then $\mathcal{B} = \mathcal{B}' \cup \{(\vec{V}W, \vec{V}W\vec{Z}) \mid (W \rightarrow \vec{V}W\vec{Z}) \in \mathcal{P}, W \text{ unnormed}\}$ is a bisimulation for \vec{X} and \vec{Y} over \mathcal{P} . \square

Correctness of the algorithm. We now focus on the correctness of the function `bisimG`. Before proceeding to soundness, we recall the *safeness property* introduced by Jančar and Moller [18].

Lemma 4 (Safeness Property). *Given a set of productions \mathcal{P} , $\vec{X} \sim_{\mathcal{P}} \vec{Y}$ if and only if the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a successful branch.*

Notice that function `bisimG` builds an expansion tree by alternating between simplification—reflexive, congruence, and BPA—and expansion operations, as proposed by Jančar and Moller. These simplification rules are *safe* [18], in the sense that the application of any rule preserves the bisimulation from a parent node to at least one child node and, reciprocally, that bisimulation on a child node implies the bisimulation of its parent node.

While the safeness property is instrumental in proving soundness, the *finite witness property* is of utmost importance to prove completeness. This result follows immediately from the analysis by Jančar and Moller [18], which capitalizes on results by Caucal [5], and Christensen, Hüttel, and Stirling [6]:

Lemma 5 (Finite Witness Property). *Given a set of productions \mathcal{P} , if $\vec{X} \sim_{\mathcal{P}} \vec{Y}$ then the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a finite successful branch.*

We refer to Caucal, Christensen, Hüttel, and Stirling for details on the proof of existence of a finite witness, as stated in Lemma 5. This proof is particularly interesting in that it highlights the importance of the BPA rules and of pruning productions on reaching such (finite) witness. The results in these two papers also elucidate the reason for the distinction, in the simplification phase, between the cases where all the symbols in the grammar are and are not normed (cf. program variable rules in function `expand`). The safeness and finite witness properties ensure the termination of the algorithm, its soundness and completeness.

Lemma 6 (Termination). *Let $(\vec{X}, \mathcal{P}') = \text{grm}(S, \emptyset)$ and $(\vec{Y}, \mathcal{P}) = \text{grm}(T, \mathcal{P}')$. Then, the computation of `bisimG`($\vec{X}, \vec{Y}, \text{prune}(\mathcal{P})$) always terminates.*

Proof. Start by noticing that `prune`(\mathcal{P}) always terminates. For `bisimG` itself, if $S \sim_{\mathcal{T}} T$ then, by Theorems 1 and 2, we have $\text{word}(S) \sim_{\text{prune}(\mathcal{P})} \text{word}(T)$ and thus the existence of a finite successful branch is ensured by the finite witness property (Lemma 5). Hence, breadth-first search eventually terminates.

When $S \not\sim_{\mathcal{T}} T$, we easily conclude that all branches in the expansion tree are finite and thus `bisimG`(\vec{X}, \vec{Y}) terminates. To conclude that all branches are finite, observe that any infinite branch is successful by definition and thus the safeness property would imply $\text{word}(S) \sim_{\text{prune}(\mathcal{P})} \text{word}(T)$ and we would have $S \sim_{\mathcal{T}} T$, by Theorems 1 and 2. \square

Lemma 7. *Let $(\vec{X}, \mathcal{P}') = \text{grm}(S, \emptyset)$ and $(\vec{Y}, \mathcal{P}) = \text{grm}(T, \mathcal{P}')$. If `bisimG`($\vec{X}, \vec{Y}, \text{prune}(\mathcal{P})$) returns **True**, then $\vec{X} \sim_{\text{prune}(\mathcal{P})} \vec{Y}$.*

Proof. Function `bisimG` returns **True** whenever it reaches a (finite) successful branch in the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$, i.e., a branch terminating in an empty node. Conclude with the safeness property, Lemma 4. \square

From the previous results, the soundness of our algorithm is now immediate: the algorithm to check the bisimulation of context-free session types is sound with respect to the meta-theory of context-free session types.

Theorem 3 (Soundness). *Let $(\vec{X}, \mathcal{P}') = \text{grm}(S, \emptyset)$ and $(\vec{Y}, \mathcal{P}) = \text{grm}(T, \mathcal{P}')$. If $\text{bisimG}(\vec{X}, \vec{Y}, \text{prune}(\mathcal{P}))$ returns **True** then $S \sim_{\mathcal{T}} T$.*

Proof. From Theorem 1, Theorem 2, and Lemma 7. □

Given that the algorithm terminates (Lemma 6), we know that if $S \not\sim_{\mathcal{T}} T$, then $\text{bisimG}(\vec{X}, \vec{Y}, \text{prune}(\mathcal{P}))$ returns **False**, where $(\vec{X}, \mathcal{P}') = \text{grm}(S, \emptyset)$ and $(\vec{Y}, \mathcal{P}) = \text{grm}(T, \mathcal{P}')$. We now show that the algorithm to check the bisimulation of context-free session types is complete with respect to the meta-theory of context-free session types. The finite witness property is paramount to achieve this result.

Theorem 4 (Completeness). *Let $(\vec{X}, \mathcal{P}') = \text{grm}(S, \emptyset)$ and $(\vec{Y}, \mathcal{P}) = \text{grm}(T, \mathcal{P}')$. If $S \sim_{\mathcal{T}} T$ then $\text{bisimG}(\vec{X}, \vec{Y}, \text{prune}(\mathcal{P}))$ returns **True**.*

Proof. Assume $S \sim_{\mathcal{T}} T$. By Theorems 1 and 2, we have $\vec{X} \sim_{\text{prune}(\mathcal{P})} \vec{Y}$. Hence, Lemma 5 ensures the existence of a finite successful branch on the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$, i.e., a branch terminating in an empty node. Since our algorithm traverses the expansion tree using breadth-first search it will, eventually, reach the empty node and conclude the bisimulation positively. □

Theorem 4 ensures that if $\text{bisimG}(\vec{X}, \vec{Y}, \mathcal{P})$ returns **False** then $S \not\sim_{\mathcal{T}} T$.

5 Evaluation

This section discusses the behaviour of our algorithm in the real world. Both for testing and for performance evaluation, we require test suites. We started with a carefully crafted, manually produced, suite of valid and invalid tests. This test suite was assembled by gathering pairs of types that emerged from examples we have studied and from programs we have written in FreeST, a programming language with context-free session types [2]. The tests produced by this method are, on the one hand, small, and, on the other hand, lacking diversity.

We then turned our attention to the automatic generation of test cases. Producing pairs of arbitrary (well-formed) types that share no variables is simple. However, the probability that a randomly generated pair of types turns out to be bisimilar is extremely low. For this reason, we generate arbitrary pairs of types that are bisimilar by construction. Theorem 5 naturally induces an algorithm: given a natural number n (the size of the pair), arbitrarily select for the base case ($n = 0$) one of the pairs in item 1 of the theorem and for the recursive case ($n \geq 1$) one of the pairs in 2–12 items.

Theorem 5 (Properties of type bisimilarity).

1. $\text{skip} \sim_{\mathcal{T}} \text{skip}$ and $\#B \sim_{\mathcal{T}} \#B$;
2. $S;T \sim_{\mathcal{T}} U;V$ if $S \sim_{\mathcal{T}} U$ and $T \sim_{\mathcal{T}} V$;
3. $\mu X.S \sim_{\mathcal{T}} \mu X.T$ if $S \sim_{\mathcal{T}} T$;
4. $\star\{\ell_i: S_i\}_{i \in I} \sim_{\mathcal{T}} \star\{\ell_i: T_i\}_{i \in I}$ if $(S_i \sim_{\mathcal{T}} T_i)_{i \in I}$;
5. $S \sim_{\mathcal{T}} T$; skip and $S \sim_{\mathcal{T}} \text{skip}$; T if $S \sim_{\mathcal{T}} T$;
6. $\star\{\ell_i: S_i\}_{i \in I}; U \sim_{\mathcal{T}} \star\{\ell_i: T_i\}_{i \in I}; V$ if $(S_i \sim_{\mathcal{T}} T_i)_{i \in I}$ and $U \sim_{\mathcal{T}} V$;
7. $T \sim_{\mathcal{T}} S$ if $S \sim_{\mathcal{T}} T$;
8. $R; (S;T) \sim_{\mathcal{T}} (U;V); W$ if $R \sim_{\mathcal{T}} U$, $S \sim_{\mathcal{T}} V$, and $T \sim_{\mathcal{T}} W$;
9. $\mu X.\mu Y.S \sim_{\mathcal{T}} \mu X.[X/Y]T \sim_{\mathcal{T}} \mu Y.[Y/X]T$ if $S \sim_{\mathcal{T}} T$;
10. $\mu X.S \sim_{\mathcal{T}} T$ if $S \sim_{\mathcal{T}} T$ and $X \notin \text{free}(S)$;
11. $[U/X]S \sim_{\mathcal{T}} [V/X]T$ if $S \sim_{\mathcal{T}} T$ and $U \sim_{\mathcal{T}} V$;
12. $\mu X.S \sim_{\mathcal{T}} [\mu X.T/X]T$ if $S \sim_{\mathcal{T}} T$.

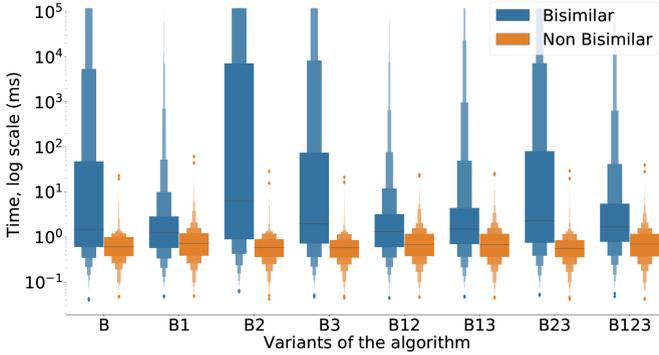
Proof. 1–3: Bisimulation is a congruence. 4–12: Thiemann and Vasconcelos [26] exhibit the appropriate bisimulations. \square

For evaluating the algorithm on non-bisimilar pairs we add the following five anti-axioms to the list in Theorem 5: (1) $\text{skip} \not\sim_{\mathcal{T}} \#B$; (2) $?B \not\sim_{\mathcal{T}} !B$; (3) $\text{skip} \not\sim_{\mathcal{T}} \star\{\ell_i: S_i\}_{i \in I}$; (4) $\oplus\{\ell_i: S_i\}_{i \in I} \not\sim_{\mathcal{T}} \&\{\ell_i: S_i\}_{i \in I}$; (5) $\star\{\ell_i: S_i\}_{i \in I} \not\sim_{\mathcal{T}} \star\{\ell_j: S_j\}_J$ where $I \subset J$. We generate two types using the same methodology as for the positive case and, then, discard the data collected when the pair turns out to be bisimilar. This produces pairs of types that are much closer than those obtained by random generation, thus hopefully approaching the reality that the compilers face when in production.

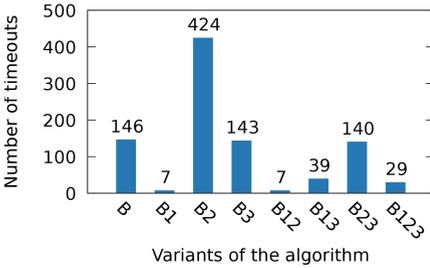
We used QuickCheck [7] to generate two test suites. That for bisimilar pairs is constructed based on Theorem 5, whereas the construction of non-bisimilar tests relies on Theorem 5 plus the anti-axioms above. Both test suites comprise 2000 entries, featuring types with a number of nodes (in the syntax tree) ranging from 1 to 200.

The base algorithm described in the previous section turns out to behave quite poorly. We then implemented the following variants.

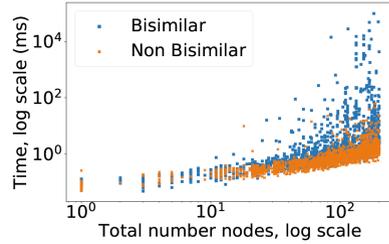
1. *Eliminating redundant productions in the grammar.* Since the size of the expansion tree depends, among other things, on the number of productions in the grammar, generating smaller grammars seems a promising optimisation. Rather than blindly adding a new production $Y \rightarrow \vec{Z}$ to the grammar (in function word, Section 3), we look, in the set of productions, for a production $W \rightarrow \vec{X}$ syntactically equal to the former, up to renaming of non-terminal symbols. In this case, we add no new production and return non-terminal W instead. To find W , we look for the least fixed-point of the transitions in the languages generated by \vec{Z} and \vec{X} and compare them. This optimisation does not compromise the results of soundness, completeness, nor termination.
2. *Using a filter rule that removes nodes with hopeless pairs.* A filter rule ensures that nodes composed by pairs of types with different norms (if normed) are removed from the expansion tree, since these types are not bisimilar. The filter rule preserves the results of soundness, completeness, and termination.



(a) Distribution of the execution time per variant



(b) Number of timeouts per variant



(c) Runtime of B1 per number of nodes

Fig. 2: Results on the test suite composed by bisimilar pairs of types is represented in blue and the test suite with non-bisimilar pairs is represented in orange. Time is in milliseconds. Scales of 2a and 2c are logarithmic; scale of 2b is linear.

3. *Using a double-ended queue to prepend promising children.* A double-ended queue allows prioritizing nodes with potential to reach an empty node faster. The algorithm prepends (rather than appends) empty nodes or nodes whose pairs (\vec{X}, \vec{Y}) are such that $|\vec{X}| \leq 1$ and $|\vec{Y}| \leq 1$. This procedure does not compromise soundness, completeness, nor termination because the number of terminal symbols is finite and the algorithm takes advantage of the reflexive and congruence rules to remove previously visited nodes from the queue.

To better understand how the algorithm performs in practice, we tested all the optimisations and their combinations. We evaluate each variant 1–3 individually (denoted by B1–B3) and all their combinations. For instance, B12 denotes the variant obtained from combining optimisations 1 and 2 above. B stands for the base algorithm, `bisimT`. We implemented the base algorithm and its variants in Haskell, using the Glasgow Haskell Compiler (version 8.6.5). The evaluation was conducted on a machine with an Intel Core i7-6700K at 4.2GHz and 8 GB of RAM running Arch Linux; tests were run under a timeout of 2 minutes.

Figure 2a depicts the distribution of the execution times (in *ms*) for both test suites and all variants. We observe that the behavior of negatives tests is roughly the same in all variants. However, the execution time for the positive tests differ from variant to variant. These differences mainly depend on the trade-off between the computational effort required for each optimisation and the efficiency they bring to deciding the equivalence of grammars. We observe that including optimisation 1 improves the execution time, while the rest, in general, does not. The combination of optimizations has a positive impact on execution time, with the exception of the B23 variant, whose distribution is worse than the base case.

Figure 2b shows the number of timeouts for each variant. The base case, B, has 146 positive tests whose execution time exceeds 2 minutes. The distribution of timeouts per variant exhibits a behavior that is consistent with that of runtime shown in Figure 2a. All combinations lead to a reduction in the number of timeouts, when compared to the base case.

Variant B1, resulting from considering optimisation 1, performs better than all others, presenting a median of 1.4 milliseconds and 7 timeouts, both for the positive tests. By taking advantage of optimisation 1, the number of timeouts reduced by 95%. The remaining positive tests take, on average, 1863.38 ms to complete with the base algorithm and 195.68 ms with variant B1, resulting in an 89% reduction in the execution time. This is the variant in production for the FreeST compiler [2].

The distribution of the execution time of B1 against the size of the input types is depicted in Figure 2c. As expected, the execution time increases considerably with the number of nodes. Although we have carried out tests with a fairly large number of nodes in the abstract syntax trees, we remark that, when used in a compiler, the algorithm will mostly come across types with a reduced number of nodes.

6 Conclusion

Context-free session types are a promising tool to describe protocols in concurrent programs. In order to be incorporated in programming languages and effectively used in compilers, a practical algorithm to decide bisimulation is called for. Taking advantage of a process algebra graph representation of types to decide bisimulation [12,13], we developed one such algorithm and proved it correct. The algorithm is incorporated in a compiler for a concurrent functional language equipped with context-free session types [2].

Possible extensions to this work include addressing higher-order session types. We also plan to extend the implementation of the algorithm to cope with context-free grammars in Greibach Normal Form that are not necessarily deterministic.

Acknowledgements. We thank Alcides Fonseca for helping with the testing process, and Filipe Casal, Alexandra Silva, and Peter Thiemman for comments and discussions. This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020 and by Cost Action CA15123 EUTypes.

References

1. Aceto, L., Hennessy, M.: Termination, deadlock, and divergence. *J. ACM* **39**(1), 147–187 (1992)
2. Almeida, B., Mordido, A., T. Vasconcelos, V.: Freest: Context-free session types in a functional language. In: *Proceedings Programming Language Approaches to Concurrency- and Communication-centric Software*. *Electronic Proceedings in Theoretical Computer Science*, vol. 291, pp. 12–23. Open Publishing Association (2019). <https://doi.org/10.4204/EPTCS.291.2>
3. Baeten, J.C., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for process generating context-free languages. *Journal of the ACM (JACM)* **40**(3), 653–682 (1993)
4. Burkart, O., Caucal, D., Steffen, B.: An elementary bisimulation decision procedure for arbitrary context-free processes. In: *Mathematical Foundations of Computer Science*. pp. 423–433 (1995). https://doi.org/10.1007/3-540-60246-1_148
5. Caucal, D.: Décidabilité de l'égalité des langages algébriques infinitaires simples. In: *Annual Symposium on Theoretical Aspects of Computer Science*. pp. 37–48. Springer (1986)
6. Christensen, S., Hüttel, H., Stirling, C.: Bisimulation equivalence is decidable for all context-free processes. *Inf. Comput.* **121**(2), 143–148 (1995)
7. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. pp. 268–279. ACM (2000), <https://doi.org/10.1145/351240.351266>
8. Czerwinski, W., Lasota, S.: Fast equivalence-checking for normed context-free processes. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2010)
9. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. *Acta Inf.* **42**(2-3), 191–225 (2005). <https://doi.org/10.1007/s00236-005-0177-z>
10. Henry, P., Sénizergues, G.: Lalblc a program testing the equivalence of dpda's. In: *International Conference on Implementation and Application of Automata*. pp. 169–180. Springer (2013)
11. Hirshfeld, Y.: Bisimulation trees and the decidability of weak bisimulations. *Electr. Notes Theor. Comput. Sci.* **5**, 2–13 (1996)
12. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theor. Comput. Sci.* **158**(1&2), 143–159 (1996). [https://doi.org/10.1016/0304-3975\(95\)00064-X](https://doi.org/10.1016/0304-3975(95)00064-X)
13. Hirshfeld, Y., Moller, F.: A fast algorithm for deciding bisimilarity of normed context-free processes. In: *CONCUR '94, Concurrency Theory*. pp. 48–63 (1994). https://doi.org/10.1007/978-3-540-48654-1_5
14. Honda, K.: Types for dyadic interaction. In: *CONCUR '93, 4th International Conference on Concurrency Theory*. LNCS, vol. 715, pp. 509–523. Springer (1993)
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: *Programming Languages and Systems*. pp. 122–138 (1998). <https://doi.org/10.1007/BFb0053567>
16. Jančar, P.: Selected ideas used for decidability and undecidability of bisimilarity. In: *International Conference on Developments in Language Theory*. pp. 56–71. Springer (2008)

17. Jancar, P.: Bisimilarity on basic process algebra is in 2-exptime (an explicit proof). arXiv preprint arXiv:1207.2479 (2012)
18. Jančar, P., Moller, F.: Techniques for decidability and undecidability of bisimilarity. In: International Conference on Concurrency Theory. pp. 30–45. Springer (1999)
19. Kiefer, S.: Bpa bisimilarity is exptime-hard. Information Processing Letters **113**(4), 101–106 (2013)
20. Padovani, L.: Context-free session type inference. In: Programming Languages and Systems - 26th European Symposium on Programming. pp. 804–830 (2017). https://doi.org/10.1007/978-3-662-54434-1_30
21. Sangiorgi, D.: An Introduction to Bisimulation and Coinduction. Cambridge University Press (2014)
22. Sénizergues, G.: The equivalence problem for deterministic pushdown automata is decidable. In: International Colloquium on Automata, Languages, and Programming. pp. 671–681. Springer (1997)
23. Stirling, C.: Decidability of DPDA equivalence. Theoretical Computer Science **255**(1-2), 1–31 (2001)
24. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994)
25. Tarski, A., et al.: A lattice-theoretical fixpoint theorem and its applications. Pacific journal of Mathematics **5**(2), 285–309 (1955)
26. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 462–475 (2016). <https://doi.org/10.1145/2951913.2951926>

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

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





Sharp Congruences Adequate with Temporal Logics Combining Weak and Strong Modalities

Frédéric Lang¹, Radu Mateescu¹, and Franco Mazzanti²

¹ Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP**, LIG, 38000 Grenoble, France
{Frederic.Lang,Radu.Mateescu}@inria.fr

² ISTI-CNR, Pisa, Italy
Franco.Mazzanti@isti.cnr.it

Abstract. We showed in a recent paper that, when verifying a modal μ -calculus formula, the actions of the system under verification can be partitioned into sets of so-called weak and strong actions, depending on the combination of weak and strong modalities occurring in the formula. In a compositional verification setting, where the system consists of processes executing in parallel, this partition allows us to decide whether each individual process can be minimized for either divergence-preserving branching (if the process contains only weak actions) or strong (otherwise) bisimilarity, while preserving the truth value of the formula. In this paper, we refine this idea by devising a family of bisimilarity relations, named sharp bisimilarities, parameterized by the set of strong actions. We show that these relations have all the nice properties necessary to be used for compositional verification, in particular congruence and adequacy with the logic. We also illustrate their practical utility on several examples and case-studies, and report about our success in the RERS 2019 model checking challenge.

Keywords: Bisimulation · Concurrency · Model checking · Mu-calculus.

1 Introduction

This paper deals with the verification of action-based, branching-time temporal properties expressible in the modal μ -calculus (L_μ) [31] on concurrent systems consisting of processes composed in parallel, usually described in languages with process algebraic flavour. A well-known problem is the state-space explosion that happens when the system state space exceeds the available computer memory.

Compositional verification is a set of techniques and tools that have proven efficient to palliate state-space explosion in many case studies [18]. They may either focus on the construction of the state space reduced for some equivalence relation, such as compositional state space construction [24, 32, 36, 43, 45–47], or on the decomposition of the full system verification into the verification of (expectedly smaller) subsystems, such as compositional reachability analysis [49, 10], assume-guarantee reasoning [41], or partial model checking [1, 34].

** Institute of Engineering Univ. Grenoble Alpes

In this paper, we focus on property-dependent compositional state space construction, where the reduction to be applied to the system is obtained by analysing the property under verification. We will refine the approach of [37] which, given a formula φ of L_μ to be verified, shows how to extract from φ a maximal hiding set of actions and a reduction (minimization for either strong [40] or divergence-preserving³ branching — divbranching for short — bisimilarity [20, 23]) that preserves the truth value of φ . The reduction is chosen according to whether φ belongs to an L_μ fragment named L_μ^{dbr} , which is adequate with divbranching bisimilarity. This fragment consists of L_μ restricted to *weak* modalities, which match actions preceded by (property-preserving) sequences of hidden actions, as opposed to traditional strong modalities $\langle\alpha\rangle\varphi_0$ and $[\alpha]\varphi_0$, which match only a single action satisfying α . If φ belongs to L_μ^{dbr} , then the system can be reduced for divbranching bisimilarity; otherwise, it can be reduced for strong bisimilarity, the weakest congruence preserving full L_μ . We call this approach of [37] the mono-bisimulation approach.

We refine the mono-bisimulation approach in [35], by handling the case of L_μ formulas containing both strong and weak modalities. To do so, fragments named $L_\mu^{strong}(A_s)$ extend L_μ^{dbr} with strong modalities matching only the actions belonging to a given set A_s of *strong* actions. This induces a partition of the parallel processes into those containing at least one strong action and those not containing any, so that a formula $\varphi \in L_\mu^{strong}(A_s)$ is still preserved if the processes containing strong actions are reduced for strong bisimilarity and the other ones for divbranching bisimilarity. We call this refined approach the combined bisimulations approach. Guidelines are also provided in [35] to extract a set of strong actions from particular L_μ formulas encoding the operators of widely-used temporal logics, such as CTL [11], ACTL [39], PDL [15], and PDL- Δ [44]. This approach is implemented on top of the CADP verification toolbox [19], and experiments show that it can improve the capabilities of compositional verification on realistic case studies, possibly reducing state spaces by orders of magnitude.

In this paper, we extend these results as follows: (1) We refine the approach by devising a family of new bisimilarity relations, called *sharp bisimilarities*, parameterized by the set of strong actions A_s . They are hybrid between strong and divbranching bisimilarities, where strong actions are handled as in strong bisimilarity whereas weak actions are handled as in divbranching bisimilarity. (2) We show that each fragment $L_\mu^{strong}(A_s)$ is adequate with the corresponding sharp bisimilarity, namely, $L_\mu^{strong}(A_s)$ is precisely the set of properties that are preserved by sharp bisimilarity (w.r.t. A_s) on all systems. (3) We show that, similarly to strong and divbranching bisimilarities, every sharp bisimilarity is a congruence for parallel composition, which enables it to be used soundly in a compositional verification setting. (4) We define an efficient state space

³ In [18, 37], the name *divergence-sensitive* is used instead of *divergence-preserving* branching bisimulation (or branching bisimulation with explicit divergences) [20, 23]. This could lead to a confusion with the relation defined in [13], also called *divergence-sensitive* but slightly different from the former relation. To be consistent in notations, we replace by *dbr* the abbreviation *dsbr* used in earlier work.

reduction algorithm that preserves sharp bisimilarity and has the same worst-case complexity as divbranching minimization. Although it is not a minimization (i.e., sharp bisimilar states may remain distinguished in the reduced state space), it coincides with divbranching minimization whenever the process it is applied to does not contain strong actions, and with strong minimization in the worst case. Therefore, applying this reduction compositionally always yields state space reduction at least as good as [35], which itself is an improvement over [37]. (5) At last, we illustrate our approach on case studies and compare our new results with those of [35, 37]. We also report about our recent success in the RERS 2019 challenge, which was obtained thanks to this new approach.

The paper is organized as follows: Sections 2 and 3 introduce the necessary background about process descriptions and temporal logic. Section 4 defines sharp bisimilarity, states its adequacy with $L_{\mu}^{strong}(A_s)$, and its congruence property for parallel composition. Section 5 presents the reduction algorithm and shows that it is correct and efficient. Section 6 illustrates our new approach on the case studies. Section 7 discusses related work. Finally, Section 8 concludes and discusses research directions for the future. The proofs of all theorems presented in this paper and a detailed description of how we tackled the RERS 2019 challenge are available in a Zenodo archive.⁴

2 Processes, Compositions, and Reductions

We consider systems of processes whose behavioural semantics can be represented using an LTS (*Labelled Transition System*).

Definition 1 (LTS). *Let \mathcal{A} be an infinite set of actions including the invisible action τ and visible actions $\mathcal{A} \setminus \{\tau\}$. An LTS P is a tuple $(\Sigma, A, \longrightarrow, p_{init})$, where Σ is a set of states, $A \subseteq \mathcal{A}$ is a set of actions, $\longrightarrow \subseteq \Sigma \times A \times \Sigma$ is the (labelled) transition relation, and $p_{init} \in \Sigma$ is the initial state. We may write $\Sigma_P, A_P, \longrightarrow_P$ for the sets of states, actions, and transitions of an LTS P , and $init(P)$ for its initial state. We assume that P is finite and write $|P|_{st}$ (resp. $|P|_{tr}$) for the number of states (resp. transitions) of P . We write $p \xrightarrow{a} p'$ for $(p, a, p') \in \longrightarrow$ and $p \xrightarrow{A}$ for $(\exists p' \in \Sigma_P, a \in A) p \xrightarrow{a} p'$.*

LTS can be composed in parallel and their actions may be abstracted away using the parallel composition and action mapping defined below, of which action hiding, cut (also known as restriction), and renaming are particular cases.

Definition 2 (Parallel composition of LTS). *Let P, Q be LTS and $A_{sync} \subseteq \mathcal{A} \setminus \{\tau\}$. The parallel composition of P and Q with synchronization on A_{sync} , written “ $P \parallel [A_{sync}] Q$ ”, is defined as $(\Sigma_P \times \Sigma_Q, A_P \cup A_Q, \longrightarrow, (init(P), init(Q)))$, where $(p, q) \xrightarrow{a} (p', q')$ if and only if (1) $p \xrightarrow{a}_P p', q' = q$, and $a \notin A_{sync}$, or (2) $p' = p, q \xrightarrow{a}_Q q'$, and $a \notin A_{sync}$, or (3) $p \xrightarrow{a}_P p', q \xrightarrow{a}_Q q'$, and $a \in A_{sync}$.*

⁴ <https://doi.org/10.5281/zenodo.3470930>

Definition 3 (Action mapping). Let P be an LTS and $\rho : A_P \rightarrow 2^A$ be a total function. We write $\rho(A_P)$ for the image of ρ , defined by $\bigcup_{a \in A_P} \rho(a)$. We write $\rho(P)$ for the LTS $(\Sigma_P, \rho(A_P), \rightarrow, \text{init}(P))$ where $\rightarrow = \{(p, a', p') \mid (\exists a \in A_P) p \xrightarrow{a} p' \wedge a' \in \rho(a)\}$. An action mapping ρ is admissible if $\tau \in A_P$ implies $\rho(\tau) = \{\tau\}$. We distinguish the following admissible action mappings:

- ρ is an action hiding if $(\exists A \subseteq \mathcal{A} \setminus \{\tau\}) (\forall a \in A \cap A_P) \rho(a) = \{\tau\} \wedge (\forall a \in A_P \setminus A) \rho(a) = \{a\}$. We write “**hide** A **in** P ” for $\rho(P)$.
- ρ is an action cut if $(\exists A \subseteq \mathcal{A} \setminus \{\tau\}) (\forall a \in A \cap A_P) \rho(a) = \emptyset \wedge (\forall a \in A_P \setminus A) \rho(a) = \{a\}$. We write “**cut** A **in** P ” for $\rho(P)$.
- ρ is an action renaming if $(\exists f : A_P \rightarrow \mathcal{A}) (\forall a \in A_P) \rho(a) = \{f(a)\}$ and $\tau \in A_P$ implies $f(\tau) = \tau$. We write “**rename** f **in** P ” for $\rho(P)$.

Parallel composition and action mapping subsume all abstraction and composition operators encodable as *networks of LTS* [42, 18, 33], such as synchronization vectors⁵ and the parallel composition, hiding, renaming, and cut operators of CCS [38], CSP [8], mCRL [26], LOTOS [29], E-LOTOS [30], and LNT [9].

LTS can be compared and reduced modulo well-known bisimilarity relations, such as strong [40] and (div)branching [20, 23] bisimilarity. We do not give their definitions, which can easily be found elsewhere (e.g., [35]). They are special cases of Definition 7 (page 7), as shown by Theorem 1 (page 9). We write \sim (resp. \sim_{dbr}) for the strong (resp. divbranching) bisimilarity relation between states. We write $\text{min}_{str}(P)$ (resp. $\text{min}_{dbr}(P)$) for the quotient of P w.r.t. strong (resp. divbranching) bisimilarity, i.e., the LTS obtained by replacing each state by its equivalence class. The quotient is the smallest LTS of its equivalence class, thus computing the quotient is called minimization. Moreover, these bisimilarities are congruences for parallel composition and admissible action mapping. This allows reductions to be applied at any intermediate step during LTS construction, thus potentially reducing the overall cost. However, since processes may constrain each other by synchronization, composing LTS pairwise following the algebraic structure of the composition expression and applying reduction after each composition can be orders of magnitude less efficient than other strategies in terms of the largest intermediate LTS. Finding an optimal strategy is impossible, as it requires to know the size of (the reachable part of) an LTS product without actually computing the product. One generally relies on heuristics to select a subset of LTS to compose at each step of LTS construction. In this paper, we will use the *smart reduction* heuristic [12, 18], which is implemented within the SVL [17] tool of CADP [19]. This heuristic tries to find an efficient composition order by analysing the synchronization and hiding structure of the composition.

⁵ For instance, the composition of P and Q where action a of P synchronizes with either b or c of Q , can be written as $\rho(P) \parallel [b, c] Q$, where ρ maps a onto $\{b, c\}$. This example illustrates the utility to map actions into sets of actions of arbitrary size.

3 Temporal Logics

Definition 4 (Modal μ -calculus [31]). *The modal μ -calculus (L_μ) is built from action formulas α and state formulas φ , whose syntax and semantics w.r.t. an LTS $P = (\Sigma, A, \longrightarrow, p_{init})$ are defined as follows:*

$$\begin{array}{l}
 \alpha ::= a \quad \llbracket a \rrbracket_A = \{a\} \\
 \quad | \text{false} \quad \llbracket \text{false} \rrbracket_A = \emptyset \\
 \quad | \alpha_1 \vee \alpha_2 \quad \llbracket \alpha_1 \vee \alpha_2 \rrbracket_A = \llbracket \alpha_1 \rrbracket_A \cup \llbracket \alpha_2 \rrbracket_A \\
 \quad | \neg\alpha_0 \quad \llbracket \neg\alpha_0 \rrbracket_A = A \setminus \llbracket \alpha_0 \rrbracket_A \\
 \\
 \varphi ::= \text{false} \quad \llbracket \text{false} \rrbracket_{P\delta} = \emptyset \\
 \quad | \varphi_1 \vee \varphi_2 \quad \llbracket \varphi_1 \vee \varphi_2 \rrbracket_{P\delta} = \llbracket \varphi_1 \rrbracket_{P\delta} \cup \llbracket \varphi_2 \rrbracket_{P\delta} \\
 \quad | \neg\varphi_0 \quad \llbracket \neg\varphi_0 \rrbracket_{P\delta} = \Sigma \setminus \llbracket \varphi_0 \rrbracket_{P\delta} \\
 \quad | \langle \alpha \rangle \varphi_0 \quad \llbracket \langle \alpha \rangle \varphi_0 \rrbracket_{P\delta} = \{p \in \Sigma \mid \exists p \xrightarrow{\alpha} p'. a \in \llbracket \alpha \rrbracket_A \wedge p' \in \llbracket \varphi_0 \rrbracket_{P\delta}\} \\
 \quad | X \quad \llbracket X \rrbracket_{P\delta} = \delta(X) \\
 \quad | \mu X. \varphi_0 \quad \llbracket \mu X. \varphi_0 \rrbracket_{P\delta} = \bigcup_{k \geq 0} \Phi_{P,\delta}^k(\emptyset)
 \end{array}$$

where $X \in \mathcal{X}$ are propositional variables denoting sets of states, $\delta : \mathcal{X} \rightarrow 2^\Sigma$ is a context mapping propositional variables to sets of states, $[\]$ is the empty context, $\delta[U/X]$ is the context identical to δ except for variable X , which is mapped to state set U , and the functional $\Phi_{P,\delta} : 2^\Sigma \rightarrow 2^\Sigma$ associated to the formula $\mu X. \varphi_0$ is defined as $\Phi_{P,\delta}(U) = \llbracket \varphi_0 \rrbracket_{P\delta}[U/X]$. For closed formulas, we write $P \models \varphi$ (read P satisfies φ) for $p_{init} \in \llbracket \varphi \rrbracket_P$.

Action formulas α are built from actions and Boolean operators. State formulas φ are built from Boolean operators, the possibility modality $\langle \alpha \rangle \varphi_0$ denoting the states with an outgoing transition labelled by an action satisfying α and leading to a state satisfying φ_0 , and the minimal fixed point operator $\mu X. \varphi_0$ denoting the least solution of the equation $X = \varphi_0$ interpreted over 2^Σ .

The usual derived operators are defined as follows: Boolean connectors $\text{true} = \neg\text{false}$ and $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$; necessity modality $[\alpha] \varphi_0 = \neg\langle \alpha \rangle \neg\varphi_0$; and maximal fixed point operator $\nu X. \varphi_0 = \neg\mu X. \neg\varphi_0[\neg X/X]$, where $\varphi_0[\neg X/X]$ is the syntactic substitution of X by $\neg X$ in φ_0 . Syntactically, $\langle \rangle$ and $[\]$ have the highest precedence, followed by \wedge , then \vee , and finally μ and ν . To have a well-defined semantics, state formulas are syntactically monotonic [31], i.e., in every subformula $\mu X. \varphi_0$, all occurrences of X in φ_0 fall in the scope of an even number of negations. Thus, negations can be eliminated by downward propagation. We now introduce the weak modalities of the fragment L_μ^{dbr} , proposed in [37].

Definition 5 (Modalities of L_μ^{dbr} [37]). *We write α_τ for an action formula such that $\tau \in \llbracket \alpha_\tau \rrbracket_A$ and α_a for an action formula such that $\tau \notin \llbracket \alpha_a \rrbracket_A$. We consider the following modalities, their L_μ semantics, and their informal semantics:*

modality name	notation	L_μ semantics
ultra-weak	$\langle (\varphi_1?.\alpha_\tau)^* \rangle \varphi_2$	$\mu X. \varphi_2 \vee (\varphi_1 \wedge \langle \alpha_\tau \rangle X)$
weak	$\langle (\varphi_1?.\alpha_\tau)^*.\varphi_1?.\alpha_a \rangle \varphi_2$	$\mu X. \varphi_1 \wedge (\langle \alpha_a \rangle \varphi_2 \vee \langle \alpha_\tau \rangle X)$
weak infinite looping	$\langle \varphi_1?.\alpha_\tau \rangle @$	$\nu X. \varphi_1 \wedge \langle \alpha_\tau \rangle X$

Ultra-weak: p is source of a path whose transition labels satisfy α_τ , leading to a state that satisfies φ_2 , while traversing only states that satisfy φ_1 .

Weak: p is source of a path whose transition labels satisfy α_τ , leading to a state that satisfies φ_1 and $\langle \alpha_a \rangle \varphi_2$, while traversing only states that satisfy φ_1 .

Weak infinite looping: p is source of an infinite path whose transition labels satisfy α_τ , while traversing only states that satisfy φ_1 .

We also consider the three dual modalities $[(\varphi_1?.\alpha_\tau)^*] \varphi_2 = \neg \langle (\varphi_1?.\alpha_\tau)^* \rangle \neg \varphi_2$, $[(\varphi_1?.\alpha_\tau)^*.\varphi_1?.\alpha_a] \varphi_2 = \neg \langle (\varphi_1?.\alpha_\tau)^*.\varphi_1?.\alpha_a \rangle \neg \varphi_2$, $[\varphi_1?.\alpha_\tau] \dashv = \neg \langle \varphi_1?.\alpha_\tau \rangle @$. The fragment L_μ^{dbr} adequate with divbranching bisimilarity consists of L_μ from which the modalities $\langle a \rangle \varphi$ and $[a] \varphi$ are replaced by the ultra-weak, weak, and weak infinite looping modalities defined above.

We identify fragments of L_μ parameterized by a set of strong actions A_s , as the set of state formulas whose action formulas contained in strong modalities satisfy only actions of A_s .

Definition 6 ($L_\mu^{strong}(A_s)$ fragment of L_μ [35]). Let $A_s \subseteq A$ be a set of actions called strong actions and α_s be any action formula such that $\llbracket \alpha_s \rrbracket_A \subseteq A_s$, called a strong action formula. $L_\mu^{strong}(A_s)$ is defined as the set of formulas semantically equivalent to some formula of the following language:

$$\begin{aligned} \varphi ::= & \text{false} \mid \varphi_1 \vee \varphi_2 \mid \neg \varphi_0 \mid \langle \alpha_s \rangle \varphi_0 \mid X \mid \mu X.\varphi_0 \\ & \mid \langle (\varphi_1?.\alpha_\tau)^* \rangle \varphi_2 \mid \langle (\varphi_1?.\alpha_\tau)^*.\varphi_1?.\alpha_a \rangle \varphi_2 \mid \langle \varphi_1?.\alpha_\tau \rangle @ \end{aligned}$$

In the context of $L_\mu^{strong}(A_s)$, we call $\langle \alpha_s \rangle \varphi_0$ a strong modality.⁶

In [35], we also provide guidelines for extracting a set A_s from particular L_μ formulas encoding the operators of widely-used temporal logics, such as CTL [11], ACTL [39], PDL [15], and PDL- Δ [44].

Example 1. The PDL formula $[\text{true}^*.a_1.a_2] \text{true}$ belongs to $L_\mu^{strong}(\{a_2\})$ as it is semantically equivalent to $[(\text{true}?.\text{true})^*.\text{true}?.a_1][a_2] \text{true}$. The CTL formula $\text{EF}(\langle a_1 \rangle \text{true} \wedge \langle a_2 \rangle \text{true})$ belongs both to $L_\mu^{strong}(\{a_1\})$ as it is semantically equivalent to $\langle (\text{true}?.\text{true})^* \rangle \langle (\langle a_1 \rangle \text{true}?.\text{true})^*.\langle a_1 \rangle \text{true}?.a_2 \rangle \text{true}$ and to $L_\mu^{strong}(\{a_2\})$ as it is semantically equivalent to the same formula where a_1 and a_2 are swapped. These formulas do not belong to $L_\mu^{strong}(\emptyset)$. (This was shown in [35].)

The latter example shows that to a formula φ may correspond several minimal sets of strong actions A_s . Indeed, either the $\langle a_1 \rangle \text{true}$ or the $\langle a_2 \rangle \text{true}$ modality can be made part of a weak modality, but not both in the same formula.

4 Sharp Bisimilarity

We define the family of sharp bisimilarity relations below. Each relation is hybrid between strong and divbranching bisimilarities, parameterized by the set of strong actions, such that the conditions of strong bisimilarity apply to strong actions and the conditions of divbranching bisimilarity apply to all other actions.

⁶ For generality we allow $\tau \in A_s$, to enable strong modalities of the form $\langle \alpha_\tau \rangle \varphi_0$.

Definition 7 (Sharp bisimilarity). A divergence-unpreserving sharp bisimulation w.r.t. a set of actions A_s is a symmetric relation $R \subseteq \Sigma \times \Sigma$ such that if $(p, q) \in R$ then for all $p \xrightarrow{a} p'$, there exists q' such that $(p', q') \in R$ and either of the following hold: (1) $q \xrightarrow{a} q'$, or (2) $a = \tau$, $\tau \notin A_s$, and $q' = q$, or (3) $a \notin A_s$, and there exists a sequence of transitions $q_0 \xrightarrow{\tau} \dots \xrightarrow{\tau} q_n \xrightarrow{a} q'$ ($n \geq 0$) such that $q_0 = q$, and for all $i \in 1..n$, $(p, q_i) \in R$.⁷ A sharp bisimulation R additionally satisfies the following divergence-preservation condition: for all $(p_0, q_0) \in R$ such that $p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \dots$ with $(p_i, q_0) \in R$ for all $i \geq 0$, there is also an infinite sequence $q_0 \xrightarrow{\tau} q_1 \xrightarrow{\tau} q_2 \xrightarrow{\tau} \dots$ such that $(p_i, q_j) \in R$ for all $i, j \geq 0$. Two states p and q are sharp bisimilar w.r.t. A_s , written $p \sim_{\sharp A_s} q$, if and only if there exists a sharp bisimulation R w.r.t. A_s such that $(p, q) \in R$.

Similarly to strong, branching, and divbranching bisimilarities, sharp bisimilarity is an equivalence relation as it is the union of all sharp bisimulations. The quotient of an LTS P w.r.t. sharp bisimilarity is unique and minimal both in number of states and number of transitions.

Example 2. Let $a, b, \omega \in \mathcal{A} \setminus \{\tau\}$, $\tau, \omega \notin A_s$. LTS P_i and P'_i of Figure 1 satisfy $P_i \sim_{\sharp A_s} P'_i$ ($i \in 1..7$). We give the smallest relation between P_i and P'_i , whose symmetric closure is a sharp bisimulation w.r.t. A_s and the weakest condition for P'_i to be minimal. Unlike divbranching, states on the same τ -cycle are not necessarily sharp bisimilar: in P'_7 , if $a \in A_s$ then p'_0 and p'_2 are not sharp bisimilar.

Example 3. The LTS of Figure 2(a) is equivalent for $\sim_{\sharp \{a\}}$ to the one of Figure 2(b), which is minimal. We see that sharp bisimilarity reduces more than strong bisimilarity when at least one action (visible or invisible) is weak. Here, τ is the only weak action and the minimized LTS is smaller than the one minimal for strong bisimilarity (only p_1 and p_2 are strongly bisimilar).

If $\tau \in A_s$, then case (2) of Definition 7 cannot apply, i.e., τ -transitions cannot be totally suppressed. As a consequence, looking at case (3), if τ -transitions are present in state q_0 then, due to symmetry, they must have a counterpart in state p . As a result, finite sequences of τ -transitions are preserved. Sharp may however differ from strong bisimilarity in the possibility to compress circuits of τ -transitions that would remain unreduced, as illustrated in Example 4 below.

Example 4. If $\tau \in A_s$ and $a \notin A_s$, then the LTS of Figure 2(b) (which is minimal for strong bisimilarity) can be reduced to the LTS of Figure 2(c).

Next theorems are new. Theorem 1 expresses that sharp bisimilarity w.r.t. a set of strong actions A_s is strictly stronger than w.r.t. any set of strong actions strictly included in A_s . Unsurprisingly, it also establishes that sharp coincides with divbranching when the set of strong actions is empty, and with strong when

⁷ We require that $(p, q_i) \in R$ for all $i \in 1..n$ and not the simpler condition $(p, q_n) \in R$ (as usual when defining branching bisimulation) because sharp bisimulation has not the nice property that $(p, q_0) \in R$ and $(p, q_n) \in R$ imply $(p, q_i) \in R$ for all $i \in 1..n$.

$a \neq \tau \wedge b \neq \tau \wedge \omega \neq \tau \wedge \tau \notin A_s \wedge \omega \notin A_s$ implies

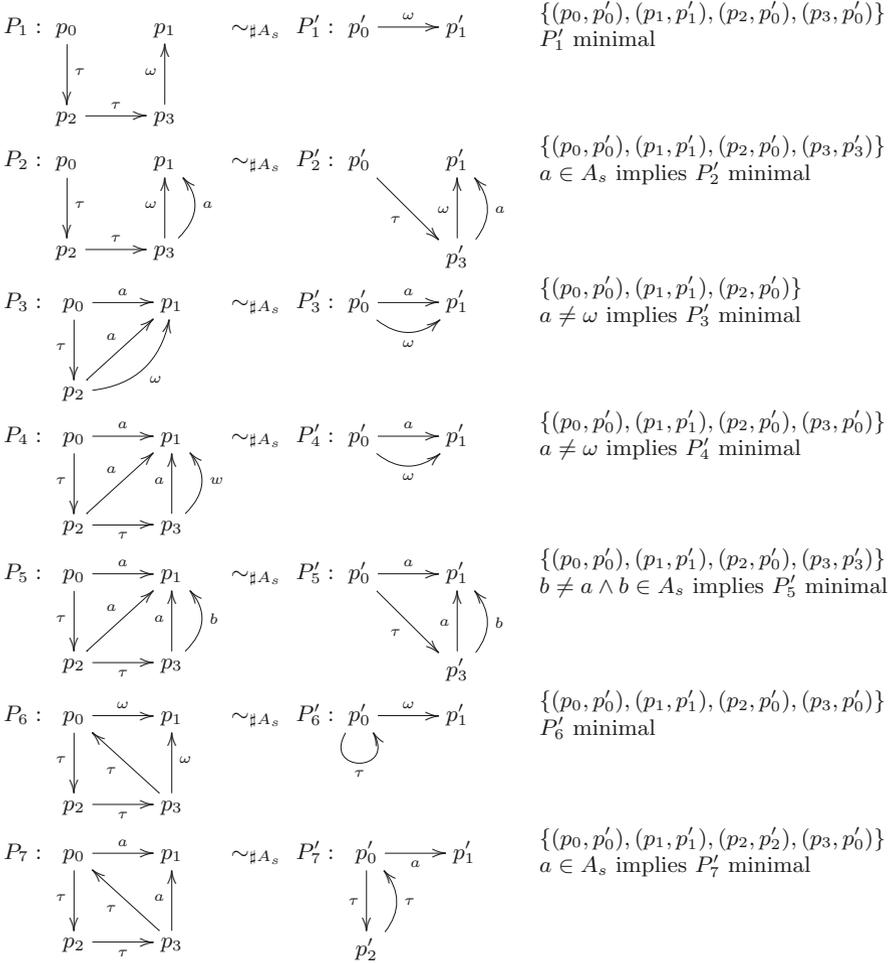


Fig. 1. Examples of sharp bisimilar LTS

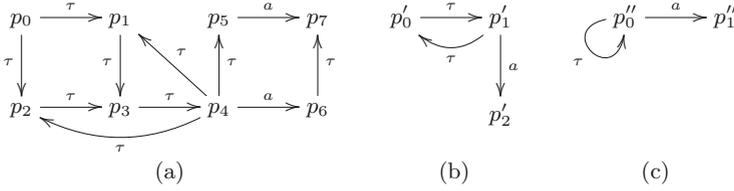


Fig. 2. LTS of Examples 3 and 4

it comprises all actions (including τ). It follows that the set of sharp bisimilarity relations equipped with set inclusion forms a complete lattice whose supremum is divbranching bisimilarity and whose infimum is strong bisimilarity.

Theorem 1. (1) $\sim_{\# \emptyset} = \sim_{dbr}$ (2) $\sim_{\# \mathcal{A}} = \sim$ (3) if $A'_s \subset A_s$ then $\sim_{\# A_s} \subset \sim_{\# A'_s}$.

Theorem 2 expresses that sharp bisimilarity w.r.t. A_s preserves the truth value of all formulas of $L_\mu^{strong}(A_s)$, and Theorem 3 that two LTS verifying exactly the same formulas of $L_\mu^{strong}(A_s)$ are sharp bisimilar. We can then deduce that $L_\mu^{strong}(A_s)$ is adequate with $\sim_{\# A_s}$, as expressed by Corollary 1.

Theorem 2. If $P \sim_{\# A_s} P'$ and $\varphi \in L_\mu^{strong}(A_s)$ then $P \models \varphi$ iff $P' \models \varphi$.

Theorem 3. If $(\forall \varphi \in L_\mu^{strong}(A_s)) P \models \varphi$ iff $Q \models \varphi$, then $P \sim_{\# A_s} Q$.

Corollary 1. $L_\mu^{strong}(A_s)$ is adequate with $\sim_{\# A_s}$, i.e., $P \sim_{\# A_s} P'$ if and only if $(\forall \varphi \in L_\mu^{strong}(A_s)) P \models \varphi$ iff $P' \models \varphi$.

Theorems 4 and 5 express that sharp bisimilarity is a congruence for parallel composition and admissible action mapping. It follows that it is also a congruence for hide, cut, and rename, as expressed by Corollary 2.

Theorem 4. If $P \sim_{\# A_s} P'$, $Q \sim_{\# A_s} Q'$ then $P \parallel [A_{sync}] Q \sim_{\# A_s} P' \parallel [A_{sync}] Q'$.

Theorem 5. If ρ is admissible and $P \sim_{\# A_s} P'$, then $\rho(P) \sim_{\# A'_s} \rho(P')$, where $A'_s = \rho(A_s) \setminus \rho(A_P \setminus A_s)$.

Corollary 2. We write A_τ for $A \cup \{\tau\}$. If $P \sim_{\# A_s} P'$ then:

- **cut** A in $P \sim_{\# A_s}$ **cut** A in P'
- **hide** A in $P \sim_{\# A_s}$ **hide** A in P' if $A_\tau \subseteq A_s \vee A_\tau \cap A_s = \emptyset$
- **rename** f in $P \sim_{\# A_s}$ **rename** f in P' if $f(A_s) \subseteq A_s \wedge f(A_P \setminus A_s) \cap A_s = \emptyset$

These theorems and corollaries generalize results on strong and divbranching bisimilarity. In particular, the side conditions of Corollary 2 are always true when $A_s = \emptyset$ (divbranching) or $A_s = \mathcal{A}$ (strong).

Since every admissible network of LTS can be translated into an equivalent composition expression consisting of parallel compositions and admissible action

mappings, Theorems 4 and 5 imply some congruence property at the level of networks of LTS. However, one must be careful on how the synchronization rules preserve or modify the set of strong actions of components.

In the sequel, we establish formally the relationship between sharp bisimilarity and sharp τ -confluence, a strong form of τ -confluence [27] defined below in a way analogous to strong τ -confluence in [28]. It is known that every τ -transition that is τ -confluent is inert for branching bisimilarity, i.e., its source and target states are branching bisimilar. There are situations where τ -confluence can be detected locally, thus enabling on-the-fly LTS reductions. We present an analogous result that might have similar applications, namely, every τ -transition that is sharp τ -confluent is inert for (divergence-unpreserving) sharp bisimilarity.

Definition 8 (Sharp τ -confluence). *Let $P = (\Sigma, A, \longrightarrow, p_{init})$ and $T \subseteq \xrightarrow{\tau}$ be a set of internal transitions. T is sharp τ -confluent w.r.t. a set A_s of strong actions if $\tau \notin A_s$ and for all $(p_0, \tau, p_1) \in T$, $a \in A$, and $p_2 \in \Sigma$: (1) $p_0 \xrightarrow{a} p_2$ implies either $p_1 \xrightarrow{a} p_2$ or there exists p_3 such that $p_1 \xrightarrow{a} p_3$ and $(p_2, \tau, p_3) \in T$, and (2) if $a \in A_s$ then $p_1 \xrightarrow{a} p_3$ implies either $p_0 \xrightarrow{a} p_3$ or there exists p_2 such that $p_1 \xrightarrow{a} p_2$ and $(p_2, \tau, p_3) \in T$. A transition $p_0 \xrightarrow{\tau} p_1$ is sharp τ -confluent w.r.t. A_s if there is a set of transitions T that is sharp τ -confluent w.r.t. A_s and such that $(p_0, \tau, p_1) \in T$.*

The difference between strong τ -confluence and sharp τ -confluence is the addition of condition (2), which can be removed to obtain the very same definition of strong τ -confluence as [28]. Strong τ -confluence thus coincides with sharp τ -confluence w.r.t. the empty set of actions. Sharp τ -confluence not only requires that other transitions of the source state of a confluent transition also exist in the target state, but also that the converse is true for strong actions.

If a transition is sharp τ -confluent w.r.t. A_s , then it is also sharp τ -confluent w.r.t. any subset of A_s . In particular, sharp τ -confluence is stronger than strong τ -confluence (which is itself stronger than τ -confluence). Theorem 6 formalizes the relationship between sharp τ -confluence and divergence-unpreserving sharp bisimilarity. This result could be lifted to sharp bisimilarity by adding a condition on divergence in the definition of sharp τ -confluence.

Theorem 6. *If $\tau \notin A_s$ and $p_0 \xrightarrow{\tau}_P p_1$ is sharp τ -confluent w.r.t. A_s , then p_0 and p_1 are divergence-unpreserving sharp bisimilar w.r.t. A_s .*

Theorem 6 illustrates a form of reduction that one can expect using sharp bisimilarity when $\tau \notin A_s$, namely compression of diamonds of sharp τ -confluent transitions, which are usually generated by parallel composition. The strongest form of sharp τ -confluence (which could be called *ultra-strong* τ -confluence) is when all visible actions are strong. In that case, every visible action present in the source state must be also present in the target state, and conversely. The source and target states are then sharp bisimilar w.r.t. the set of visible actions. Yet, it is interesting to note that they are not necessarily strongly bisimilar, sharp bisimilarity w.r.t. all visible actions being weaker than strong bisimilarity.

There exist weaker forms of τ -confluence [27, 50], which accept that choices between τ -confluent and other transitions are closed by arbitrary sequences of τ -confluent transitions rather than sequences of length 0 or 1. It could be interesting to investigate how the definition of sharp τ -confluence could also be weakened, while preserving inertness for sharp bisimilarity.

5 LTS Reduction

The interest of sharp bisimilarity in the context of compositional verification is the ability to replace components by smaller but still equivalent ones, as allowed by the congruence property. To do so, we need a procedure that enables such a reduction. This is what we address in this section.

A procedure to reduce an LTS P for sharp bisimilarity is proposed as follows: (1) Build P' , consisting of P in which all τ -transitions that immediately precede a transition labelled by a strong action (or all τ -transitions if τ is itself a strong action) are renamed into a special visible action $\kappa \in \mathcal{A} \setminus A_P$; (2) Minimize P' for divbranching bisimilarity; (3) Hide in the resulting LTS all occurrences of κ . The renaming of τ -transitions into κ allows them to be considered temporarily as visible transitions, so that they are not eliminated by divbranching minimization.⁸ This algorithm is now defined formally.

Definition 9. *Let P be an LTS and A_s be a set of strong actions. Let $\kappa \in \mathcal{A} \setminus A_P$ be a special visible action. We write $red_{A_s}(P)$ for the reduction of P defined as the LTS “hide κ in $min_{dbr}(P')$ ”, where $P' = (\Sigma_P, A_P \cup \{\kappa\}, \longrightarrow, init(P))$ and \longrightarrow is defined as follows:*

$$\begin{aligned} \longrightarrow = & \{(p, \kappa, p') \mid p \xrightarrow{a}_{\rightarrow P} p' \wedge \underline{\kappa}(a, p')\} \cup \{(p, a, p') \mid p \xrightarrow{a}_{\rightarrow P} p' \wedge \neg \underline{\kappa}(a, p')\} \\ \text{where } \underline{\kappa}(a, p') = & ((a = \tau) \wedge (\tau \in A_s \vee p' \xrightarrow{A_s}_{\rightarrow P})) \end{aligned}$$

It is clear that $red_{A_s}(P)$ is a reduction, i.e., it cannot have more states and transitions than P . Since the complexities of the transformation from P to P' and of hiding κ are at worst linear in $|P|_{tr}$, the complexity of the whole algorithm is dominated by divbranching minimization, for which there exists an algorithm⁹ of worst-case complexity $\mathcal{O}(m \log n)$, where $m = |P|_{tr}$ and $n = |P|_{st}$ [25].

As regards correctness, Theorem 7 states that $red_{A_s}(P)$ is indeed sharp bisimilar to P . Theorem 8 indicates that the reduction coincides with divbranching minimization if the LTS does not contain any strong action, with strong minimization if τ is a strong action or if the LTS does not contain τ , and that the resulting LTS has a size that lies in between the size of the minimal LTS for divbranching bisimilarity and the size of the minimal LTS for strong bisimilarity.

Theorem 7. *For any LTS P , we have $P \sim_{\sharp A_s} red_{A_s}(P)$.*

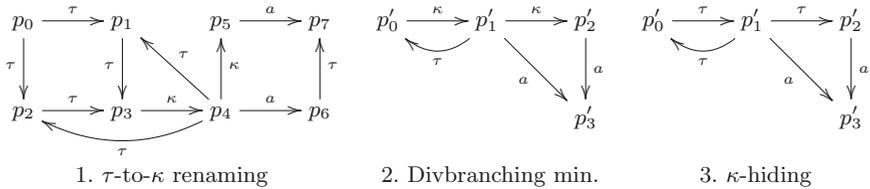
⁸ The letter κ stands for *keep uncompressed*.

⁹ Strictly speaking, the algorithm of [25] implements branching minimization but, as noted by its authors, handling divergences requires only a minor adaptation.

Theorem 8. *The following hold for any LTS P : (1) if $A_P \cap A_s = \emptyset$ then $red_{A_s}(P) = min_{dbr}(P)$, (2) if $\tau \notin A_P \setminus A_s$ then $red_{A_s}(P) = min_{str}(P)$, and (3) $|min_{dbr}(P)|_{st} \leq |red_{A_s}(P)|_{st} \leq |min_{str}(P)|_{st} \wedge |min_{dbr}(P)|_{tr} \leq |red_{A_s}(P)|_{tr} \leq |min_{str}(P)|_{tr}$.*

Although sharp reduction is effective in practice, as will be illustrated in the next section, it may fail to compress τ -transitions that are inert for sharp bisimilarity, as show the following examples.

Example 5. Consider the LTS of Figure 2(a) (page 9). Its reduction using the above algorithm consists of the three steps depicted below:



The reduced LTS (obtained at step 3) has one more state and two more transitions than the minimal LTS shown in Figure 2(b). Even though all visible actions are strong, our reduction compresses more than strong bisimilarity (recall that the minimal LTS for strong bisimilarity has 7 states and 8 transitions). In general, our reduction reduces more than strong bisimilarity¹⁰ as soon as $\tau \notin A_s$ (which is the case for most formulas in practice).

Example 6. In Figure 1 (page 8), if $a \in A_s$ then $red_{A_s}(P_1) = P'_1$, $red_{A_s}(P_2) = P'_2$, and $red_{A_s}(P_6) = P'_6$, i.e., reduction yields the minimal LTS. Yet, $red_{A_s}(P_3) = P_3 \neq P'_3$, i.e., the sharp τ -confluent transition $p_0 \xrightarrow{\tau} p_3$ p_2 is not compressed. Similarly, P_4 , P_5 , and P_7 are not minimized using red_{A_s} .

Devising a minimization algorithm for sharp bisimilarity is left for future work. It could combine elements of existing partition-refinement algorithms for strong and divbranching minimizations, but the following difficulty must be taken into account (basic knowledge about partition-refinement is assumed):

- A sequence of τ -transitions is inert w.r.t. the current state partition if both its source, target, and intermediate states are in the same block. To refine a partition for sharp bisimilarity, one must be able to compute efficiently the set of non-inert transitions labelled by weak actions and reachable after an arbitrary sequence of inert transitions. The potential presence of inert cycles has to be considered carefully to avoid useless computations.

¹⁰ The result of reduction is necessarily strong-bisimulation minimal, because if a transition $p \xrightarrow{\tau} p'$ is renamed into κ , then it is also the case of a τ -transition in every state bisimilar to p , which remains bisimilar after the renaming. In addition, the subsequent divbranching minimization step necessarily merges strongly bisimilar states.

- In the case of divbranching bisimilarity, every τ -cycle is inert and can thus be compressed into a single state. This is usually done initially, using the Tarjan algorithm for finding strongly connected components, whose complexity is linear in the LTS size. This guarantees the absence of inert cycles (except self τ -loops) all along the subsequent partition-refinement steps. However, τ -cycles are not necessarily inert for sharp bisimilarity, as illustrated by LTS P'_7 in Figure 1 (page 8). Therefore, τ -cycles cannot be compressed initially. Instead, a cycle inert w.r.t. the current partition may be split into several sub-blocks during a refinement step. To know whether the sub-blocks still contain inert cycles, the Tarjan algorithm may have to be applied again.

Although red_{A_s} is not a minimization, we will see that it performs very well when used in a compositional setting. The reason is that (1) only a few of the system actions are strong, which limits the number of τ -transitions renamed to κ , and (2) sharp τ -confluent transitions most often originate from the interleaving of τ -transitions that are inert in the components of parallel composition. The above reduction algorithm removes most inert transitions in individual (sequential) LTS, thus limiting the number of sharp τ -confluent transitions in intermediate LTS. Still, better reductions can be expected with a full minimization algorithm, which will compress all τ -transitions that are inert for sharp bisimilarity.

6 Experimentation

We experimented sharp reduction on the examples presented in [35] (consisting of formulas containing both weak and strong modalities), namely the TFTP (Trivial File Transfer Protocol) and the CTL verification problems on parallel systems of the RERS 2018 challenge. For lack of space, see [35] for more details about these case studies. In both cases, we composed parallel processes in the same order as we did using the combined bisimulations approach, but using sharp bisimilarity instead of strong or divbranching bisimilarity to reduce processes. Experiments were done on a 3GHz/12GB RAM/8-core Intel Xeon computer running Linux, using the specification languages and 32-bit versions of tools provided in the CADP toolbox version 2019-d “Pisa” [19].

The results are given in Figures 3 (TFTP) and 4 (RERS 2018), both in terms of the size of the largest intermediate LTS, the size of the final LTS (LTS obtained after the last reduction step, on which the formula is checked), memory consumption, and time. Each subfigure contains three curves corresponding to the mono-bisimulation approach (using strong bisimulation to reduce all LTS), the combined bisimulations approach, and the sharp bisimulation approach. The former two curves are made from data that were already presented in [35]. Note that the vertical axis of all subfigures is on a logarithmic scale. In the RERS 2018 case, the mono-bisimulation approach gives results only for experiments 101#22 and 101#23, all other experiments failing due to state space explosion.¹¹

¹¹ E.g., smart mono-bisimulation fails on problem 103#23 after generating an intermediate LTS with more than 4.5 billion states and 36 billion transitions (instead of 50,301 states and 334,530 transitions using sharp bisimulation) using Grid’5000 [6].

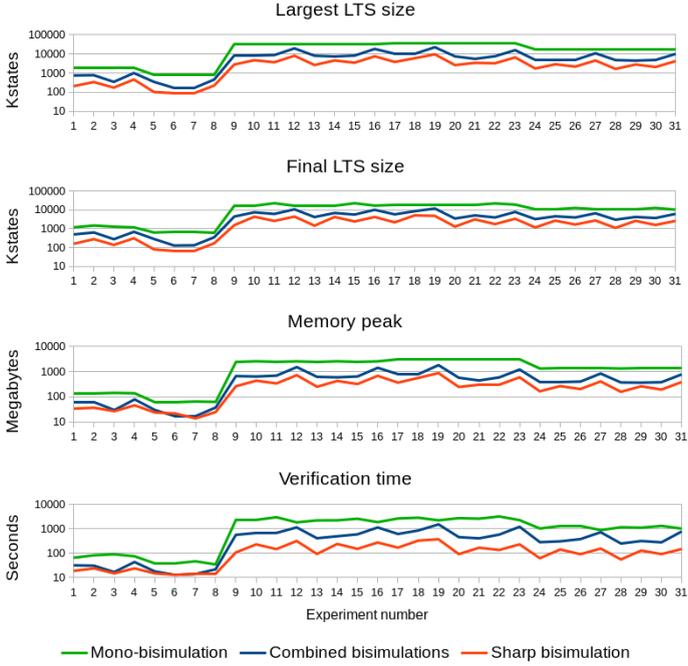


Fig. 3. Experimental results of the TFTP case-study

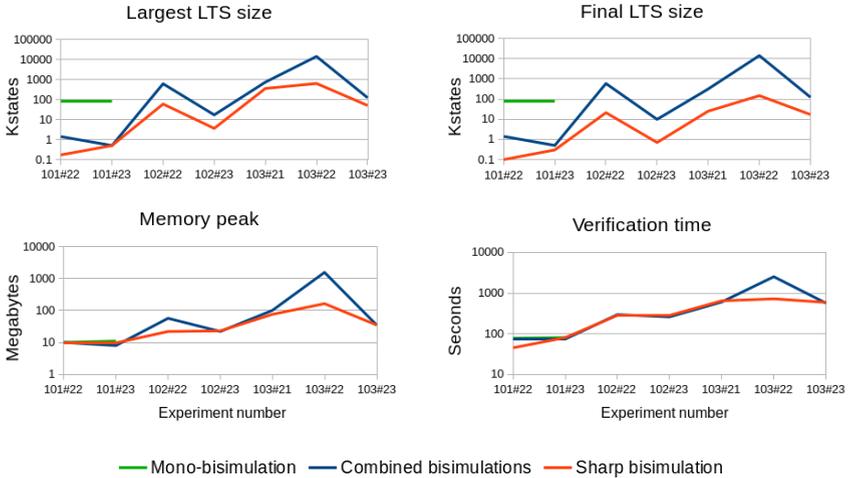


Fig. 4. Experimental results of the RERS 2018 case-study

These results show that sharp bisimilarity incurs much more LTS reduction than the combined bisimulations approach, by a factor close to the one obtained when switching from the mono-bisimulation approach to the combined bisimulations approach. However, in the case of the RERS 2018 examples, this gain on LTS size does not always apply to time and/or memory consumption in the same proportions, except for experiment 103#22. This suggests that our implementation of minimization could be improved.

These experiments were conducted after closing of the RERS 2018 challenge. Encouraged by the good results obtained with these two approaches, we participated to the 2019 edition¹², where 180 CTL problems were proposed instead of 9 in 2018. The models on which the properties had to be verified have from 8 to 70 parallel processes and from 29 to 234 actions. Although the models had been given in a wealth of different input formats (communicating automata, Petri nets in PNML format with NUPN information [16], and Promela) suitable for a large number of model checking tools, no other team than ours participated to the parallel challenges. This is a significant difference with 2018, when the challenge was easier, allowing three teams (with different tools) to participate.

We applied smart sharp reduction to these problems, using a prototype program that extracts strong actions automatically from (a restricted set of) CTL formulas used in the competition.¹³ This allowed the 180 properties to be checked automatically in less than 2.5 hours (CPU time), and using about 200 MB of RAM only, whereas using strong reduction failed on most of the largest problems. The largest intermediate graph obtained for the whole set of problems has 3364 states. All results were correct and we won all gold medals¹⁴ in this category.¹⁵ Details are available in the Zenodo archive mentioned in the introduction.

7 Related Work

The paper [48] defines on doubly-labelled transition systems (mix between Kripke structure and LTS) a family of bisimilarity relations derived from divbranching bisimilarity, parameterized by a natural number n , which preserves CTL* formulas whose nesting of next operators is smaller or equal to n . Similar to our work, they show that this family of relations (which is distinct from sharp bisimilarity in that there is no distinction between weak and strong actions) fills the gap between strong and divbranching bisimilarities. They apply their bisimilarity relation to slicing rather than compositional verification.

The paper [2] proposes that, if the formula contains only so-called *selective* modalities, of the form $\langle\langle(-\alpha_1)^*.\alpha_2\rangle\rangle\varphi_0$, then all actions but those satisfying

¹² <http://rers-challenge.org/2019>

¹³ The paper [35] presents identities that were used to extract such strong actions.

¹⁴ A RERS gold medal is not a ranking but an achievement, not weakened by the low number of competitors. We also won all gold medals in the “verification of LTL properties on parallel systems” category, using an adaptation of this approach.

¹⁵ <http://cadp.inria.fr/news12.html>

α_1 or α_2 can be hidden, and the resulting system can be reduced for $\tau^*.a$ -equivalence [14]. Yet, there exist formulas whose strong modalities $\langle\alpha\rangle\varphi_0$ cannot translate into anything but the selective modality $\langle(-\text{true})^*\alpha\rangle\varphi$, meaning that no action at all can be hidden. In this case, $\tau^*.a$ equivalence coincides with strong bisimilarity and thus incurs much less reduction than sharp bisimilarity. Moreover, it is well-known that $\tau^*.a$ -equivalence is not a congruence for parallel composition [7], which makes it unsuitable to compositional verification, even to check formulas that contain weak modalities only.

The adequacy of L_μ^{dbr} with divbranching bisimilarity is shown in [37]. This paper also claims that $\text{ACTL}\setminus X$ is as expressive as L_μ^{dbr} and thus also adequate with divbranching bisimilarity, but a small mistake in the proof had the authors omit that the L_μ^{dbr} formula $\langle\tau\rangle@$ cannot actually be expressed in $\text{ACTL}\setminus X$. It remains true that $\text{ACTL}\setminus X$ is preserved by divbranching bisimilarity.

In [13], it is shown that $\text{ACTL}\setminus X$ is adequate with divergence sensitive branching bisimilarity. This bisimilarity relation is equivalent to divbranching bisimilarity [21–23] only in the case of deadlock-free LTS, but it differs in the presence of deadlock states since it does not distinguish a deadlock state from a self τ -loop (which can instead be recognized in L_μ^{dbr} with the $\langle\tau\rangle@$ formula).

8 Conclusion

This work enhances the reductions that can be obtained by combining compositional LTS construction with an analysis of the temporal logic formula to be verified. In particular, known results about strong and divbranching bisimilarities have been combined into a new family of relations called sharp bisimilarities, which inherit all nice properties of their ancestors and refine the state of the art in compositional verification.

This new approach is promising. Yet, to be both usable by non-experts and fully efficient, at least two components are still missing: (1) The sets of strong actions, which are a key ingredient in the success of this approach, still have to be computed either using pencil and paper or using tools dedicated to restricted logics; automating their computation in the case of arbitrary L_μ formulas is not easy, but likely feasible, opening the way to a new research track; finding a minimal set of strong actions automatically is challenging, and since it is not unique, even more challenging is the quest for the set that will incur the best reductions. (2) Efficient algorithms are needed to minimize LTS for sharp bisimilarity; they could probably be obtained by adapting the known algorithms for strong and divbranching minimizations (at least using some kind of signature-based partition refinement algorithm in the style of Blom *et al.* [3–5] in a first step), but this remains to be done.

Acknowledgements. The authors thank Hubert Garavel, who triggered our collaboration between Grenoble and Pisa, and Wendelin Serwe for his comments on earlier versions of this paper. They also thank the anonymous referees for the pertinence of their comments, which allowed significant improvements of this paper.

References

1. Andersen, H.R.: Partial model checking. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science LICS (San Diego, California, USA). pp. 398–407. IEEE Computer Society Press (Jun 1995)
2. Barbuti, R., De Francesco, N., Santone, A., Vaglini, G.: Selective mu-calculus and formula-based equivalence of transition systems. *Journal of Computer and System Sciences* **59**, 537–556 (1999)
3. Blom, S., Orzan, S.: A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. *Software Tools for Technology Transfer* **7**(1), 74–86 (2005)
4. Blom, S., Orzan, S.: Distributed State Space Minimization. *Software Tools for Technology Transfer* **7**(3), 280–291 (2005)
5. Blom, S., van de Pol, J.: Distributed branching bisimulation minimization by inductive signatures. In: Proceedings of the 8th International Workshop on Parallel and Distributed Methods in verification PDMC 2009 (Eindhoven, The Netherlands). *Electronic Proceedings in Theoretical Computer Science*, vol. 14 (2009)
6. Bolze, R., Cappello, F., Caron, E., Daydé, M.J., Desprez, F., Jéannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quétier, B., Richard, O., Talbi, E., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *IJHPCA* **20**(4), 481–494 (2006). <https://doi.org/10.1177/1094342006070078>, <https://doi.org/10.1177/1094342006070078>
7. Bouajjani, A., Fernandez, J.C., Graf, S., Rodríguez, C., Sifakis, J.: Safety for branching time semantics. In: Proceedings of 18th ICALP. Springer (Jul 1991)
8. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. *J. ACM* **31**(3), 560–599 (Jul 1984)
9. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LNT to LOTOS Translator (Version 6.7) (Jul 2017), INRIA, Grenoble, France
10. Cheung, S.C., Kramer, J.: Enhancing Compositional Reachability Analysis with Context Constraints. In: Proceedings of the 1st ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Los Angeles, CA, USA). pp. 115–125. ACM Press (Dec 1993)
11. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* **8**(2), 244–263 (Apr 1986)
12. Crouzen, P., Lang, F.: Smart Reduction. In: Giannakopoulou, D., Orejas, F. (eds.) Proceedings of Fundamental Approaches to Software Engineering (FASE'11), Saarbrücken, Germany. *Lecture Notes in Computer Science*, vol. 6603, pp. 111–126. Springer (Mar 2011)
13. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. *Journal of the Association for Computing Machinery* (1990)
14. Fernandez, J.C., Mounier, L.: “On the Fly” Verification of Behavioural Equivalences and Preorders. In: Larsen, K.G., Skou, A. (eds.) Proceedings of the 3rd Workshop on Computer-Aided Verification (CAV'91), Aalborg, Denmark. *Lecture Notes in Computer Science*, vol. 575, pp. 181–191. Springer (Jul 1991)
15. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.* **18**(2), 194–211 (Sep 1979)
16. Garavel, H.: Nested-Unit Petri Nets. *Journal of Logical and Algebraic Methods in Programming* **104**, 60–85 (Apr 2019)

17. Garavel, H., Lang, F.: SVL: a Scripting Language for Compositional Verification. In: Kim, M., Chin, B., Kang, S., Lee, D. (eds.) Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea. pp. 377–392. Kluwer Academic Publishers (Aug 2001), full version available as INRIA Research Report RR-4223
18. Garavel, H., Lang, F., Mateescu, R.: Compositional Verification of Asynchronous Concurrent Systems Using CADP. *Acta Informatica* **52**(4), 337–392 (Apr 2015)
19. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer (STTT)* **15**(2), 89–107 (Apr 2013)
20. van Glabbeek, R.J., Weijland, W.P.: Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam (1989), also in proc. IFIP 11th World Computer Congress, San Francisco, 1989
21. van Glabbeek, R.J., Luttik, B., Trcka, N.: Branching bisimilarity with explicit divergence. *Fundam. Inform.* **93**(4), 371–392 (2009). <https://doi.org/10.3233/FI-2009-109>, <https://doi.org/10.3233/FI-2009-109>
22. van Glabbeek, R.J., Luttik, B., Trcka, N.: Computation tree logic with deadlock detection. *Logical Methods in Computer Science* **5**(4) (2009), <http://arxiv.org/abs/0912.2109>
23. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM* **43**(3), 555–600 (1996)
24. Graf, S., Steffen, B.: Compositional Minimization of Finite State Systems. In: Clarke, E.M., Kurshan, R.P. (eds.) Proceedings of the 2nd Workshop on Computer-Aided Verification (CAV'90), Rutgers, New Jersey, USA. Lecture Notes in Computer Science, vol. 531, pp. 186–196. Springer (Jun 1990)
25. Groote, J.F., Jansen, D.N., Keiren, J.J.A., Wijs, A.: An $o(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. *ACM Transactions on Computational Logic* **18**(2) (2017)
26. Groote, J., Ponse, A.: The Syntax and Semantics of μ CRL. CS-R 9076, Centrum voor Wiskunde en Informatica, Amsterdam (1990)
27. Groote, J.F., Sellink, M.P.A.: Confluence for process verification. *Theoretical Computer Science* **170**(1–2), 47–81 (1996)
28. Groote, J., Pol, J.: State space reduction using partial τ -confluence. In: Nielsen, M., Rovan, B. (eds.) Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'00), Bratislava, Slovakia. Lecture Notes in Computer Science, vol. 1893, pp. 383–393. Springer (Aug 2000), also available as CWI Technical Report SEN-R0008, Amsterdam, March 2000
29. ISO/IEC: LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva (Sep 1989)
30. ISO/IEC: Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization – Information Technology, Geneva (Sep 2001)
31. Kozen, D.: Results on the propositional μ -calculus. *Theoretical Computer Science* **27**, 333–354 (1983)
32. Krimm, J.P., Mounier, L.: Compositional State Space Generation from LOTOS Programs. In: Brinksma, E. (ed.) Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97),

- University of Twente, Enschede, The Netherlands. Lecture Notes in Computer Science, vol. 1217. Springer (Apr 1997), extended version with proofs available as Research Report VERIMAG RR97-01
33. 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.) Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'05), Eindhoven, The Netherlands. Lecture Notes in Computer Science, vol. 3771, pp. 70–88. Springer (Nov 2005), full version available as INRIA Research Report RR-5673
 34. Lang, F., Mateescu, R.: Partial Model Checking using Networks of Labelled Transition Systems and Boolean Equation Systems. *Logical Methods in Computer Science* **9**(4), 1–32 (Oct 2013)
 35. Lang, F., Mateescu, R., Mazzanti, F.: Compositional verification of concurrent systems by combining bisimulations. In: McIver, A., ter Beek, M. (eds.) Proceedings of the 23rd International Symposium on Formal Methods – 3rd World Congress on Formal Methods FM 2019 (Porto, Portugal). Lecture Notes in Computer Science, vol. 11800, pp. 196–213. Springer (2019)
 36. Malhotra, J., Smolka, S.A., Giacalone, A., Shapiro, R.: A Tool for Hierarchical Design and Simulation of Concurrent Systems. In: Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems, Stirling, Scotland, UK. pp. 140–152. British Computer Society (Jul 1988)
 37. Mateescu, R., Wijs, A.: Property-Dependent Reductions Adequate with Divergence-Sensitive Branching Bisimilarity. *Sci. Comput. Program.* **96**(3), 354–376 (2014)
 38. Milner, R.: *Communication and Concurrency*. Prentice-Hall (1989)
 39. Nicola, R.D., Vaandrager, F.W.: Action versus State based Logics for Transition Systems, Lecture Notes in Computer Science, vol. 469, pp. 407–419. Springer (Apr 1990)
 40. Park, D.: Concurrency and Automata on Infinite Sequences. In: Deussen, P. (ed.) *Theoretical Computer Science*. Lecture Notes in Computer Science, vol. 104, pp. 167–183. Springer (Mar 1981)
 41. Pnueli, A.: In transition from global to modular temporal reasoning about programs. *Logic and Models of Concurrent Systems* **13**, 123–144 (1984)
 42. de Putter, S., Wijs, A., Lang, F.: Compositional model checking is lively — extended version (2019), submitted to *Science of Computer Programming*
 43. Sabnani, K.K., Lapone, A.M., Ümit Uyar, M.: An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications* **37**(9), 940–948 (Sep 1989)
 44. Streett, R.: Propositional dynamic logic of looping and converse. *Information and Control* (54), 121–141 (1982)
 45. Tai, K.C., Koppol, P.V.: An Incremental Approach to Reachability Analysis of Distributed Programs. In: Proceedings of the 7th International Workshop on Software Specification and Design, Los Angeles, CA, USA. pp. 141–150. IEEE Press, Piscataway, NJ (Dec 1993)
 46. Tai, K.C., Koppol, P.V.: Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In: Proceedings of the IEEE International Conference on Network Protocols, San Francisco, CA, USA. pp. 318–325. IEEE Press, Piscataway, NJ (Oct 1993)
 47. Valmari, A.: Compositional State Space Generation. In: Rozenberg, G. (ed.) *Advances in Petri Nets 1993 – Papers from the 12th International Conference on*

- Applications and Theory of Petri Nets (ICATPN'91), Gjern, Denmark. Lecture Notes in Computer Science, vol. 674, pp. 427–457. Springer (1993)
48. Yatapanage, N., Winter, K.: Next-preserving branching bisimulation. *Theoretical Computer Science* **594**, 120–142 (2015)
 49. Yeh, W.J., Young, M.: Compositional Reachability Analysis Using Process Algebra. In: *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis, and Verification (SIGSOFT'91)*, Victoria, British Columbia, Canada. pp. 49–59. ACM Press (Oct 1991)
 50. Ying, M.: Weak confluence and τ -inertness. *Theoretical Computer Science* **238**, 465–475 (2000)

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

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



Verification and Efficiency



How Many Bits Does it Take to Quantize Your Neural Network?

Mirco Giacobbe^{1,2}, Thomas A. Henzinger¹, and Mathias Lechner¹

¹ IST Austria, Klosterneuburg, Austria

² University of Oxford, Oxford, United Kingdom

Abstract. Quantization converts neural networks into low-bit fixed-point computations which can be carried out by efficient integer-only hardware, and is standard practice for the deployment of neural networks on real-time embedded devices. However, like their real-numbered counterpart, quantized networks are not immune to malicious misclassification caused by adversarial attacks. We investigate how quantization affects a network's robustness to adversarial attacks, which is a formal verification question. We show that neither robustness nor non-robustness are monotonic with changing the number of bits for the representation and, also, neither are preserved by quantization from a real-numbered network. For this reason, we introduce a verification method for quantized neural networks which, using SMT solving over bit-vectors, accounts for their exact, bit-precise semantics. We built a tool and analyzed the effect of quantization on a classifier for the MNIST dataset. We demonstrate that, compared to our method, existing methods for the analysis of real-numbered networks often derive false conclusions about their quantizations, both when determining robustness and when detecting attacks, and that existing methods for quantized networks often miss attacks. Furthermore, we applied our method beyond robustness, showing how the number of bits in quantization enlarges the gender bias of a predictor for students' grades.

1 Introduction

Deep neural networks are powerful machine learning models, and are becoming increasingly popular in software development. Since recent years, they have pervaded our lives: think about the language recognition system of a voice assistant, the computer vision employed in face recognition or self driving, not to talk about many decision-making tasks that are hidden under the hood. However, this also subjects them to the resource limits that real-time embedded devices impose. Mainly, the requirements are low energy consumption, as they often run on batteries, and low latency, both to maintain user engagement and to effectively interact with the physical world. This translates into specializing our computation by reducing the memory footprint and instruction set, to minimize cache misses and avoid costly hardware operations. For this purpose, quantization compresses neural networks, which are traditionally run over 32-bit

floating-point arithmetic, into computations that require bit-wise and integer-only arithmetic over small words, e.g., 8 bits. Quantization is the standard technique for the deployment of neural networks on mobile and embedded devices, and is implemented in TensorFlow Lite [13]. In this work, we investigate the robustness of quantized networks to adversarial attacks and, more generally, formal verification questions for quantized neural networks.

Adversarial attacks are a well-known vulnerability of neural networks [24]. For instance, a self-driving car can be tricked into confusing a stop sign with a speed limit sign [9], or a home automation system can be commanded to deactivate the security camera by a voice reciting poetry [22]. The attack is carried out by superposing the innocuous input with a crafted perturbation that is imperceptible to humans. Formally, the attack lies within the neighborhood of a known-to-be-innocuous input, according to some notion of distance. The fraction of samples (from a large set of test inputs) that do not admit attacks determines the robustness of the network. We ask ourselves how quantization affects a network’s robustness or, dually, how many bits it takes to ensure robustness above some specific threshold. This amounts to proving that, for a set of given quantizations and inputs, there does not exist an attack, which is a formal verification question.

The formal verification of neural networks has been addressed either by overapproximating—as happens in abstract interpretation—the space of outputs given a space of attacks, or by searching—as it happens in SMT-solving—for a variable assignment that witnesses an attack. The first category includes methods that relax the neural networks into computations over interval arithmetic [20], treat them as hybrid automata [27], or abstract them directly by using zonotopes, polyhedra [10], or tailored abstract domains [23]. Overapproximation-based methods are typically fast, but incomplete: they prove robustness but do not produce attacks. On the other hand, methods based on local gradient descent have turned out to be effective in producing attacks in many cases [16], but sacrifice formal completeness. Indeed, the search for adversarial attack is NP-complete even for the simplest (i.e., ReLU) networks [14], which motivates the rise of methods based on *Satisfiability Modulo Theory* (SMT) and *Mixed Integer Linear Programming* (MILP). SMT-solvers have been shown not to scale beyond toy examples (20 hidden neurons) on monolithic encodings [21], but today’s specialized techniques can handle real-life benchmarks such as, neural networks for the MNIST dataset. Specialized tools include DLV [12], which subdivides the problem into smaller SMT instances, and Planet [8], which combines different SAT and LP relaxations. Reluplex takes a step further augmenting LP-solving with a custom calculus for ReLU networks [14]. At the other end of the spectrum, a recent MILP formulation turned out effective using off-the-shelf solvers [25]. Moreover, it formed the basis for Sherlock [7], which couples local search and MILP, and for a specialized branch and bound algorithm [4].

All techniques mentioned above do not reason about the machine-precise semantics of the networks, neither over floating- nor over fixed-point arithmetic, but reason about a real-number relaxation. Unfortunately, adversarial attacks

computed over the reals are not necessarily attacks on execution architectures, in particular, for quantized networks implementations. We show, for the first time, that attacks and, more generally, robustness and vulnerability to attacks do not always transfer between real and quantized networks, and also do not always transfer monotonically with the number of bits across quantized networks. Verifying the real-valued relaxation of a network may lead scenarios where

- (i) specifications are fulfilled by the real-valued network but not for its quantized implementation (false negative),
- (ii) specifications are violated by the real-valued network but fulfilled by its quantized representation (false negatives), or
- (iii) counterexamples witnessing that the real-valued network violated the specification, but do not witness a violation for the quantized network (invalid counterexamples/attacks).

More generally, we show that all three phenomena can occur non-monotonically with the precision in the numerical representation. In other words, it may occur that a quantized network fulfills a specification while both a higher and a lower bits quantization violate it, or that the first violates it and both the higher and lower bits quantizations fulfill it; moreover, specific counterexamples may not transfer monotonically across quantizations.

The verification of real-numbered neural networks using the available methods is inadequate for the analysis of their quantized implementations, and the analysis of quantized neural networks needs techniques that account for their bit-precise semantics. Recently, a similar problem has been addressed for binarized neural networks, through SAT-solving [18]. Binarized networks represent the special case of 1-bit quantizations. For many-bit quantizations, a method based on gradient descent has been introduced recently [28]. While efficient (and sound), this method is incomplete and may produce false negatives.

We introduce, for the first time, a complete method for the formal verification of quantized neural networks. Our method accounts for the bit-precise semantics of quantized networks by leveraging the first-order theory of bit vectors without quantifiers (QF_BV), to exactly encode hardware operations such as 2'complementation, bit-shift, integer arithmetic with overflow. On the technical side, we present a novel encoding which balances the layout of long sequences of hardware multiply-add operations occurring in quantized neural networks. As a result, we obtain an encoding into a first-order logic formula which, in contrast to a standard unbalanced linear encoding, makes the verification of quantized networks practical and amenable to modern bit-precise SMT-solving. We built a tool using Boolector [19], evaluated the performance of our encoding, compared its effectiveness against real-numbered verification and gradient descent for quantized networks, and finally assessed the effect of quantization for different networks and verification questions.

We measured the robustness to attacks of a neural classifier involving 890 neurons and trained on the MNIST dataset (handwritten digits), for quantizations between 6 and 10 bits. First, we demonstrated that Boolector, off-the-shelf and using our balanced SMT encoding, can compute every attack within 16

hours, with a median time of 3h 41m, while timed-out on all instances beyond 6 bits using a standard linear encoding. Second, we experimentally confirmed that both Reluplex and gradient descent for quantized networks can produce false conclusions about quantized networks; in particular, spurious results occurred consistently more frequently as the number of bits in quantization decreases. Finally, we discovered that, to achieve an acceptable level of robustness, it takes a higher bit quantization than is assessed by standard accuracy measures.

Lastly, we applied our method beyond the property of robustness. We also evaluate the effect of quantization upon the gender bias emerging from quantized predictors for students’ performance in mathematics exams. More precisely, we computed the maximum predictable grade gap between any two students with identical features except for gender. The experiment showed that a substantial gap existed and was proportionally enlarged by quantization: the lower the number bits the larger the gap.

We summarize our contribution in five points. First, we show that the robustness of quantized neural networks is non-monotonic in the number of bits and is non-transferable from the robustness of their real-numbered counterparts. Second, we introduce the first complete method for the verification of quantized neural networks. Third, we demonstrate that our encoding, in contrast to standard encodings, enabled the state-of-the-art SMT-solver Boolector to verify quantized networks with hundreds of neurons. Fourth, we also show that existing methods determine both robustness and vulnerability of quantized networks less accurately than our bit-precise approach, in particular for low-bit quantizations. Fifth, we illustrate how quantization affects the robustness of neural networks, not only with respect to adversarial attacks, but also with respect to other verification questions, specifically fairness in machine learning.

2 Quantization of Feed-forward Networks

A feed-forward neural network consists of a finite set of *neurons* x_1, \dots, x_k partitioned into a sequence of layers: an *input layer* with n neurons, followed by one or many *hidden layers*, finally followed by an *output layer* with m neurons. Every pair of neurons x_j and x_i in respectively subsequent layers is associated with a *weight* coefficient $w_{ij} \in \mathbb{R}$; if the layer of x_j is not subsequent to that of x_i , then we assume $w_{ij} = 0$. Every hidden or output neuron x_i is associated with a *bias* coefficient $b_i \in \mathbb{R}$. The real-valued semantics of the neural network gives to each neuron a real value: upon a valuation for the neurons in the input layer, every other neuron x_i assumes its value according to the update rule

$$x_i = \text{ReLU-}N(b_i + \sum_{j=1}^k w_{ij}x_j), \quad (1)$$

where $\text{ReLU-}N: \mathbb{R} \rightarrow \mathbb{R}$ is the *activation function*. Altogether, the neural network implements a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ whose result corresponds to the valuation for the neurons in the output layer.

The activation function governs the firing logic of the neurons, layer by layer, by introducing non-linearity in the system. Among the most popular activation functions are purely non-linear functions, such as the tangent hyperbolic and the sigmoidal function, and piece-wise linear functions, better known as *Rectified Linear Units* (ReLU) [17]. ReLU consists of the function that takes the positive part of its argument, i.e., $\text{ReLU}(x) = \max\{x, 0\}$. We consider the variant of ReLU that imposes a cap value N , known as ReLU- N [15]. Precisely

$$\text{ReLU-}N(x) = \min\{\max\{x, 0\}, N\}, \tag{2}$$

which can be alternatively seen as a concatenation of two ReLU functions (see Eq. 10). As a consequence, all neural networks we treat are full-fledged ReLU networks; their real-valued versions are amenable to state-of-the-art verification tools including Reluplex, but neither account for the exact floating- nor fixed-point execution models.

Quantizing consists of converting a neural network over real numbers, which is normally deployed on floating-point architectures, into a neural network over integers, whose semantics corresponds to a computation over fixed-point arithmetic [13]. Specifically, fixed-point arithmetic can be carried out by integer-only architectures and possibly over small words, e.g., 8 bits. All numbers are represented in 2's complement over B bits words and F bits are reserved to the fractional part: we call the result a *B-bits quantization in QF arithmetic*. More concretely, the conversion follows from the rounding of weight and bias coefficients to the F -th digit, namely $\bar{b}_i = \text{rnd}(2^F b_i)$ and $\bar{w}_{ij} = \text{rnd}(2^F w_{ij})$ where $\text{rnd}(\cdot)$ stands for any rounding to an integer. Then, the fundamental relation between a quantized value \bar{a} and its real counterpart a is

$$a \approx 2^{-F} \bar{a}. \tag{3}$$

Consequently, the semantics of a quantized neural network corresponds to the update rule in Eq. 1 after substituting of x , w , and b with the respective approximants $2^{-F} \bar{x}$, $2^{-F} \bar{w}$, and $2^{-F} \bar{b}$. Namely, the semantics amounts to

$$\bar{x}_i = \text{ReLU}(2^F N)(\bar{b}_i + \text{int}(2^{-F} \sum_{j=1}^k \bar{w}_{ij} \bar{x}_j)), \tag{4}$$

where $\text{int}(\cdot)$ truncates the fractional part of its argument or, in other words, rounds towards zero. In summary, the update rule for the quantized semantics consists of four parts. The first part, i.e., the linear combination $\sum_{j=1}^k \bar{w}_{ij} \bar{x}_j$, propagates all neurons values from the previous layer, obtaining a value with possibly $2B$ fractional bits. The second scales the result by 2^{-F} truncating the fractional part by, in practice, applying an arithmetic shift to the right of F bits. Finally, the third applies the bias \bar{b} and the fourth clamps the result between 0 and $2^F N$. As a result, a quantize neural network realizes a function $f: \mathbb{Z}^n \rightarrow \mathbb{Z}^m$, which exactly represents the concrete (integer-only) hardware execution.

We assume all intermediate values, e.g., of the linear combination, to be fully representable as, coherently with the common execution platforms [13], we

always allocate enough bits for under and overflow not to happen. Hence, any loss of precision from the respective real-numbered network happens exclusively, at each layer, as a consequence of rounding the result of the linear combination to F fractional bits. Notably, rounding causes the robustness to adversarial attacks of quantized networks with different quantization levels to be independent of one another, and independent of their real counterpart.

3 Robustness is Non-monotonic in the Number of Bits

A neural classifier is a neural network that maps a n -dimensional input to one out of m classes, each of which is identified by the output neuron with the largest value, i.e., for the output values z_1, \dots, z_m , the choice is given by

$$\text{class}(z_1, \dots, z_m) = \arg \max_i z_i. \quad (5)$$

For example, a classifier for handwritten digits takes in input the pixels of an image and returns 10 outputs z_0, \dots, z_9 , where the largest indicates the digit the image represents. An adversarial attack is a perturbation for a sample input

$$\text{original} + \text{perturbation} = \text{attack}$$

that, according to some notion of closeness, is indistinguishable from the original, but tricks the classifier into inferring an incorrect class. The attack in Fig. 1 is



Fig. 1: Adversarial attack.

indistinguishable from the original by the human eye, but induces our classifier to assign the largest value to z_3 , rather than z_9 , misclassifying the digit as a 3. For this example, misclassification happens consistently, both on the real-numbered and on the respective 8-bits quantized network in Q4 arithmetic. Unfortunately, attacks do not necessarily transfer between real and quantized networks and neither between quantized networks for different precision. More generally, attacks and, dually, robustness to attacks are non-monotonic with the number of bits.

We give a prototypical example for the non-monotonicity of quantized networks in Fig. 2. The network consists of one input, 4 hidden, and 2 output neurons from left to right. Weights and bias coefficients, which are annotated on the edges, are all fully representable in Q1. For the neurons in the top row we show, respectively from top to bottom, the valuations obtained using a Q3, Q2, and Q1 quantization of the network (following Eq. 4); precisely, we

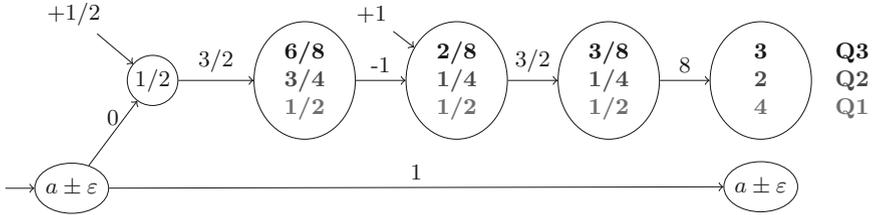


Fig. 2: Neural network with non-monotonic robustness w.r.t. its Q1, Q2, and Q3 quantizations.

show their fractional counterpart $\bar{x}/2^F$. We evaluate all quantizations and obtain that the valuations for the top output neuron are non-monotonic with the number of fractional bits; in fact, the Q1 dominates the Q3 which dominates the Q2 output. Coincidentally, the valuations for the Q3 quantization correspond to the valuations with real-number precision (i.e., never undergo truncation), indicating that also real and quantized networks are similarly incomparable. Notably, all phenomena occur both for quantized networks with rounding towards zero (as we show in the example), and with rounding to the nearest, which is naturally non-monotonic (e.g., $5/16$ rounds to $1/2$, $1/4$, and $3/8$ with, resp., Q1, Q2, and Q3).

Non-monotonicity of the output causes non-monotonicity of robustness, as we can put the decision boundary of the classifier so as to put Q2 into a different class than Q1 and Q3. Suppose the original sample is $3/2$ and its class is associated with the output neuron on the top, and suppose attacks can only lay in the neighboring interval $3/2 \pm 1$. In this case, we obtain that the Q2 network admits an attack, because the bottom output neuron can take $5/2$, that is larger than 2. On the other hand, the bottom output can never exceed $3/8$ and $1/2$, hence Q1 and Q3 are robust. Dually, also non-robustness is non-monotonic as, for the sample $9/2$ whose class corresponds to the bottom neuron, for the interval $9/2 \pm 2$, Q2 is robust while both Q3 and Q1 are vulnerable. Notably, the specific attacks of Q3 and Q1 also do not always coincide as, for instance, $7/2$.

Robustness and non-robustness are non-monotonic in the number of bits for quantized networks. As a consequence, verifying a high-bits quantization, or a real-valued network, may derive false conclusions about a target lower-bits quantization, in either direction. Specifically, for the question as for whether an attack exists, we may have both (i) false negatives, i.e., the verified network is robust but the target network admits an attack, and (ii) false positives, i.e., the verified network is vulnerable while the target network robust. In addition, we may also have (iii) true positives with invalid attacks, i.e., both are vulnerable but the found attack do not transfer to the target network. For these reasons we introduce a verification method quantized neural network that accounts for their bit-precise semantics.

4 Verification of Quantized Networks using Bit-precise SMT-solving

Bit-precise SMT-solving comprises various technologies for deciding the satisfiability of first-order logic formulae, whose variables are interpreted as bit-vectors of fixed size. In particular, it produces satisfying assignments (if any exist) for formulae that include bitwise and arithmetic operators, whose semantics corresponds to that of hardware architectures. For instance, we can encode bit-shifts, 2’s complementation, multiplication and addition with overflow, signed and unsigned comparisons. More precisely, this is the quantifier-free first-order theory of bit-vectors (i.e., QF_BV), which we employ to produce a monolithic encoding of the verification problem for quantized neural networks.

A verification problem for the neural networks f_1, \dots, f_K consists of checking the validity of a statement of the form

$$\varphi(\mathbf{y}_1, \dots, \mathbf{y}_K) \implies \psi(f_1(\mathbf{y}_1), \dots, f_K(\mathbf{y}_K)), \quad (6)$$

where φ is a predicate over the inputs and ψ over the outputs of all networks; in other words, it consists of checking an input–output relation, which generalizes various verification questions, including robustness to adversarial attacks and fairness in machine learning, which we treat in Sec. 5. For the purpose of SMT solving, we encode the verification problem in Eq. 6, which is a validity question, by its dual satisfiability question

$$\varphi(\mathbf{y}_1, \dots, \mathbf{y}_K) \wedge \bigwedge_{i=1}^K f_i(\mathbf{y}_i) = \mathbf{z}_i \wedge \neg\psi(\mathbf{z}_1, \dots, \mathbf{z}_K), \quad (7)$$

whose satisfying assignments constitute counterexamples for the contract. The formula consists of three conjuncts: the rightmost constrains the input within the assumption, the leftmost forces the output to violate the guarantee, while the one in the middle relates inputs and outputs by the semantics of the neural networks.

The semantics of the network consists of the bit-level translation of the update rule in Eq. 4 over all neurons, which we encode in the formula

$$\bigwedge_{i=1}^k x_i = \text{ReLU}(2^F N)(x'_i) \wedge x'_i = \bar{b}_i + \text{ashr}(x''_i, F) \wedge x''_i = \sum_{j=1}^k \bar{w}_{ij} x_j. \quad (8)$$

Each conjunct in the formula employs three variables x , x' , and x'' and is made of three, respective, parts. The first part accounts for the operation of clamping between 0 and $2^F N$, whose semantics is given by the formula $\text{ReLU-}M(x) = \text{ite}(\text{sign}(x), 0, \text{ite}(x \geq M, M, x))$. Then, the second part accounts for the operations of scaling and biasing. In particular, it encodes the operation of rounding by truncation scaling, i.e., $\text{int}(2^{-F}x)$, as an arithmetic shift to the right. Finally, the last part accounts for the propagation of values from the previous layer, which, despite the obvious optimization of pruning away all monomials

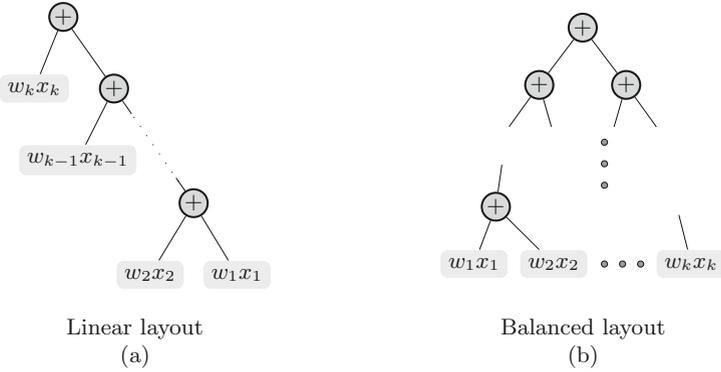


Fig. 3: Abstract syntax trees for alternative encodings of a long linear combination of the form $\sum_{i=1}^k w_i x_i$.

with null coefficient, often consists of long linear combinations, whose exact semantic amounts to a sequence of multiply-add operations over an accumulator; particularly, encoding it requires care in choosing variables size and association layout.

The size of the bit-vector variables determines whether overflows can occur. In particular, since every monomial $w_{ij}x_j$ consists of the multiplication of two B -bits variables, its result requires $2B$ bits in the worst case; since summation increases the value linearly, its result requires a logarithmic amount of extra bits in the number of summands (regardless of the layout). Provided that, we avoid overflow by using variables of $2B + \log k$ bits, where k is the number of summands.

The association layout is not unique and, more precisely, varies with the order of construction of the long summation. For instance, associating from left to right produces a linear layout, as in Fig. 3a. Long linear combinations occurring in quantized neural networks are implemented as sequences of multiply-add operations over a single accumulator; this naturally induces a linear encoding. Instead, for the purpose formal verification, we propose a novel encoding which re-associates the linear combination by recursively splitting the sum into equal parts, producing a *balanced layout* as in Fig. 3b. While linear and balanced layouts are semantically equivalent, we have observed that, in practice, the second impacted positively the performance of the SMT-solver as we discuss in Sec. 5, where we also compare against other methods and investigate different verification questions.

5 Experimental Results

We set up an experimental evaluation benchmark based on the MNIST dataset to answer the following three questions. First, how does our balanced encoding

scheme impact the runtime of different SMT solvers compared to a standard linear encoding? Then, how often can robustness properties, that are proven for the real-valued network, transferred to the quantized network and vice versa? Finally, how often do gradient based attacking procedures miss attacks for quantized networks?

The MNIST dataset is a well-studied computer vision benchmark, which consists of 70,000 handwritten digits represented by 28-by-28 pixel images with a single 8-bit grayscale channel. Each sample belongs to exactly one category $\{0, 1, \dots, 9\}$, which a machine learning model must predict from the raw pixel values. The MNIST set is split into 60,000 training and 10,000 test samples.

We trained a neural network classifier on MNIST, following a *post-training quantization* scheme [13]. First, we trained, using TensorFlow with floating-point precision, a network composed of 784 inputs, 2 hidden layers of size 64, 32 with ReLU-7 activation function and 10 outputs, for a total of 890 neurons. The classifier yielded a *standard accuracy*, i.e., the ratio of samples that are correctly classified out of all samples in the testing set, of 94.7% on the floating-point architecture. Afterward, we quantized the network with various bit sizes, with the exception of imposing the input layer to be always quantized in 8 bits, i.e., the original precision of the samples. The quantized networks required at least Q3 with 7 total bits to obtain an accuracy above 90% and Q5 with 10 bits to reach 94%. For this reason, we focused our study on the quantizations from 6 and the 10 bits in, respectively, Q2 to Q6 arithmetic.

Robust accuracy or, more simply, *robustness* measure the ratio of robust samples: for the distance $\varepsilon > 0$, a sample a is robust when, for all its perturbations y within that distance, the classifier class $\circ f$ chooses the original class $c = \text{class} \circ f(a)$. In other words, a is robust if, for all y

$$|a - y|_\infty \leq \varepsilon \implies c = \text{class} \circ f(y), \quad (9)$$

where, in particular, the right-hand side can be encoded as $\bigwedge_{j=1}^m z_j \leq z_c$, for $z = f(y)$. Robustness is a validity question as in Eq. 6 and any witness for the dual satisfiability question constitutes an adversarial attack. We checked the robustness of our selected networks over the first 300 test samples from the dataset with $\varepsilon = 1$ on the first 200 and $\varepsilon = 2$ on the next 100; in particular, we tested our encoding using the SMT-solver Boolector [19], Z3 [5], and CVC4 [3], off-the-shelf.

Our experiments serve two purposes. The first is evaluating the scalability and precision of our approach. As for scalability, we study how encoding layout, i.e., linear or balanced, and the number of bits affect the runtime of the SMT-solver. As for precision, we measured the gap between our method and both a formal verifier for real-numbered networks, i.e., Reluplex [14], and the IFGSM algorithm [28], with respect to the accuracy of identifying robust and vulnerable samples. The second purpose of our experiments is evaluating the effect of quantization on the robustness to attacks of our MNIST classifier and, with an additional experiment, measuring the effect of quantization over the gender fairness of a student grades predictor, also demonstrating the expressiveness of our method beyond adversarial attacks.

As we only compared the verification outcomes, any complete verifier for real-numbered networks would lead to the same results as those obtained with Reluplex. Note that these tools verify the real-numbered abstraction of the network using some form of linear real arithmetic reasoning. Consequently, rounding errors introduced by the floating-point implementation of both, the network and the verifier, are not taken into account.

5.1 Scalability and performance

We evaluated whether our balanced encoding strategy, compared to a standard linear encoding, can improve the scalability of contemporary SMT solvers for quantifier-free bit-vectors (QF_BV) to check specifications of quantized neural networks. We ran all our experiments on an Intel Xeon W-2175 CPU, with 64GB memory, 128GB swap file, and 16 hours of time budget per problem instance. We encoded each instance using the two variants, the standard linear and our balanced layout. We scheduled 14 solver instances in parallel, i.e., the number of physical processor cores available on our machine. While Z3, CVC4 and Yices2

SMT-solver	Encoding	6-bit	7-bit	8-bit	9-bit	10-bit
Boolector [19]	Linear (standard)	3h 25m	oot	oot	oot	oot
	Balanced (ours)	18m	1h 29m	3h 41m	5h 34m	8h 58m
Z3 [5]	Linear (standard)	oot	-	-	-	-
	Balanced (ours)	oot	-	-	-	-
CVC4 [3]	Linear (standard)	oom	-	-	-	-
	Balanced (ours)	oom	-	-	-	-
Yices2 [6]	Linear (standard)	oot	-	-	-	-
	Balanced (ours)	oot	-	-	-	-

Table 1: Median runtimes for bit-exact robustness checks. The term oot refers to timeouts, and oom refers to out-of-memory errors. Due to the poor performance of Z3, CVC4, and Yices2 on our smallest 6-bit network, we abstained from running experiments involving more than 6 bits, i.e., entries marked by a dash (-).

timed out or ran out of memory on the 6-bit network, Boolector could check the instances of our smallest network within the given time budget, independently of the employed encoding scheme. Our results align with the SMT-solver performances reported by the SMT-COMP 2019 competition in the QF_BV division [11]. Consequently, we will focus our discussion on the results obtained with Boolector.

With linear layout Boolector timed-out on all instances but the smallest networks (6 bits), while with the balanced layout it checked all instances with an overall median runtime of 3h 41m and, as shown in Tab. 1, roughly doubling at every bits increase, as also confirmed by the histogram in Fig. 4.

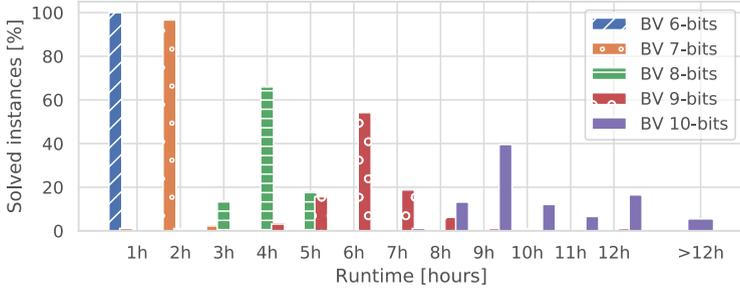


Fig. 4: Runtimes for bit-exact adversarial robustness checks of a classifier trained on the MNIST dataset using Boolector and our balanced SMT encodings. Runtime roughly doubles with each additional bit used for the quantization.

Our results demonstrate that our balanced association layout improves the performance of the SMT-solver, enabling it to scale to networks beyond 6 bits. Conversely, a standard linear encoding turned out to be ineffective on all tested SMT solvers. Besides, our method tackled networks with 890 neurons which, while small compared to state-of-the-art image classification models, already pose challenging benchmarks for the formal verification task. In the real-numbered world, for instance, off-the-shelf solvers could initially tackle up to 20 neurons [20], and modern techniques, while faster, are often evaluated on networks below 1000 neurons [14,4].

Additionally, we pushed our method to its limits, refining our MNIST network to a four-layers deep Convolutional network (2 Conv + 2 Fully-connected layers) with a total of 2238 neurons, which achieved a test accuracy of 98.56%. While for the 6-bits quantization we proved robustness for 99% of the tested samples within a median runtime of 3h 39min, for 7-bits and above all instances timed-out. Notably, Reluplex also failed on the real-numbered version, reporting numerical instability.

5.2 Comparison to other methods

Looking at existing methods for verification, one has two options to verify quantized neural networks: verifying the real-valued network and hoping the functional property is preserved when quantizing the network, or relying on incomplete methods and hoping no counterexample is missed. A question that emerges is how accurate are these two approaches for verifying robustness of a quantized network? To answer this question, we used Reluplex [14] to prove the robustness of the real-valued network. Additionally, we compared to the Iterative Fast Gradient Sign Method (IFGSM), which has recently been proposed to generate ℓ_∞ -bounded adversarial attacks for quantized networks [28]; notably, IFGSM is

incomplete in the sense that it may miss attacks. We then compared these two verification outcomes to the ground-truth obtained by our approach.

In our study, we employ the following notation. We use the term "false negative" (i) to describe cases in which the quantized network can be attacked, while no attack exists that fools the real-number network. Conversely, the term "false positive" (ii) describes the cases in which a real-number attack exists while the quantized network is robust. Furthermore, we use the term "invalid attack" (iii) to specify attacks produced for the real-valued network that fools the real-valued network but not the quantized network.

Regarding the real-numbered encoding, Reluplex accepts only pure ReLU networks. For this reason, we translate our ReLU- N networks into functionally equivalent ReLU networks, by translating each layer with

$$\text{ReLU-}N(W \cdot \mathbf{x} + \mathbf{b}) = \text{ReLU} \left(-I \cdot \text{ReLU}(-W \cdot \mathbf{x} - \mathbf{b} + N) \right). \quad (10)$$

Out of the 300 samples, at least one method timed out on 56 samples, leaving us with 244 samples whose results were computed over all networks. Tab. 2 depicts how frequently the robustness property could be transferred from the real-valued network to the quantized networks. Not surprisingly, we observed the trend that when increasing the precision of the network, the error between the quantized model and the real-valued model decreases. However, even for the 10-bit model, in 0.8% of the tested samples, verifying the real-valued model leads to a wrong conclusion about the robustness of the quantized network. Moreover, our results show the existence of samples where the 10-bit network is robustness while the real-valued is attackable and vice versa. The invalid attacks illustrate that the higher the precision of the quantization, the more targeted attacks need to be. For instance, while 94% of attacks generated for the real-valued network represented valid attacks on the 7-bit model, this percentage decrease to 80% for the 10-bit network.

Bits	True negatives	False negatives (i)	False positives (ii)	True positives	Invalid attacks (iii)
6	66.4%	25.0%	3.3%	5.3%	8%
7	84.8%	6.6%	1.6%	7.0%	6%
8	88.5%	2.9%	0.4%	8.2%	10%
9	91.0%	0.4%	0.4%	8.2%	20%
10	91.0%	0.4%	0.4%	8.2%	20%

Table 2: Transferability of vulnerability from the verification outcome of the real-valued network to the verification outcome of the quantized model. While vulnerability is transferable between the real-valued and the higher precision networks, (9 and 10-bits), in most of the tested cases, this discrepancy significantly increases when compressing the networks with fewer bits, i.e. see columns (i) and (ii).

Next, we compared how well incomplete methods are suited to reason about the robustness of quantized neural networks. We employed IFGSM to attack the 244 test samples for which we obtained the ground-truth robustness and measure how often IFGSM is correct about assessing the robustness of the network. For the sake of completeness, we perform the same analysis for the real-valued network.

Bits	True negatives	False negatives (i)	False positives (ii)	True positives
6	69.7%	1.2 %	-	30.3%
7	86.5%	1.6 %	-	13.5%
8	88.9%	0.8 %	-	11.1%
9	91.4%	0.8 %	-	8.6 %
10	91.4%	0 %	-	8.6 %
\mathbb{R}	91.4%	0 %	-	8.6 %

Table 3: Transferability of incomplete robustness verification (IFGSM [28]) to ground-truth robustness (ours) for quantized networks. While for the real-valued and 10-bit networks our gradient based incomplete verification did not miss any possible attack, a non-trivial number of vulnerabilities were missed by IFGSM for the low-bit networks. The row indicted by \mathbb{R} compares IFGSM attacking the floating-point implementation to the ground-truth obtained, using Reluplex, by verifying the real-valued relaxation of the network.

Our results in Tab. 3 present the trend that with higher precision, e.g., 10-bits or reals, incomplete methods provide a stable estimate about the robustness of the network, i.e., IFGSM was able to find attacks for all non-robust samples. However, for lower precision levels, IFGSM missed a substantial amount of attacks, i.e., for the 7-bit network, IFGSM could not find a valid attack for 10% of the non-robust samples.

5.3 The effect of quantization on robustness

In Tab. 3 we show how standard accuracy and robust accuracy degrade on our MNIST classifier when increasing the compression level. The data indicates a constant discrepancy between standard accuracy and robustness; for real numbered networks, a similar fact was already known in the literature [26]: we empirically confirm that observation for our quantized networks, whose discrepancy fluctuated between 3 and 4% across all precision levels. Besides, while an acceptable, larger than 90%, standard accuracy was achieved at 7 bits, an equally acceptable robustness was achieved at 9 bits.

One relationship not shown in Tab. 3 is that these 4% of non-robust samples are not equal for across quantization levels. For instance, we observed samples

Precision	6	7	8	9	10	\mathbb{R}
Standard	73.4%	91.8%	92.2%	94.3%	95.5%	94.7%
Robust	69.7%	86.5%	88.9%	91.4%	91.4%	91.4%

Table 4: Accuracy of the MNIST classifiers on the 244 test samples for which all quantization levels could be checked within the given time budget. The column indicated by \mathbb{R} compares the accuracy of the floating-point implementation to the robust accuracy of the real-valued relaxation of the network.

that are robust for 7-bit network but attackable when quantizing with 9- and 10-bits. Conversely, there are attacks for the 7-bit networks that are robust samples in the 8-bit network.

5.4 Network specifications beyond robustness

Concerns have been raised that decisions of an ML system could discriminate towards certain groups due to a bias in the training data [2]. A vital issue in quantifying fairness is that neural networks are black-boxes, which makes it hard to explain how each input contributes to a particular decision.

We trained a network on a publicly available dataset consisting of 1000 students’ personal information and academic test scores [1]. The personal features include gender, parental level of education, lunch plans, and whether the student took a preparation course for the test, all of which are discrete variables. We train a predictor for students’ math scores, which is a discrete variable between 0 and 100. Notably, the dataset contains a potential source for gender bias: the mean math score among females is 63.63, while it is 68.73 among males.

The network we trained is composed of 2 hidden layers with 64 and 32 units, respectively. We use a 7-bit quantization-aware training scheme, achieving a 4.14% mean absolute error, i.e., the difference between predicted and actual math scores on the test set.

The network is *fair* if the gender of a person influences the predicted math score by at most the bias β . In other words, checking fairness amounts to verifying that

$$\bigwedge_{i \neq \text{gender}} s_i = t_i \wedge s_{\text{gender}} \neq t_{\text{gender}} \implies |f(\mathbf{s}) - f(\mathbf{t})| \leq \beta, \quad (11)$$

is valid over the variables \mathbf{s} and \mathbf{t} , which respectively model two students for which gender differs but all other features are identical—we call them twin students. When we encode the dual formula, we encode two copies of the semantics of the same network: to one copy we give one student \mathbf{s} and take the respective grade g , to the other we give its twin \mathbf{t} and take grade h ; precisely, we check for the unsatisfiability the negation of formula in Eq. 11. Then, we compute a tight upper bound for the bias, that is the maximum possible change in predicted score for any two twins. To compute the tightest bias, we progressively increase β until our encoded formula becomes unsatisfiable.

We measure mean test error and gender bias of the 6- to the 10-bits quantization of the networks. We show the results in Tab. 5. The test error was stable

Quantization level	Mean test error	Tightest bias upper bound
6 bits	4.46	22
7 bits	4.14	17
8 bits	4.37	16
9 bits	4.38	15
10 bits	4.59	15

Table 5: Results for the formal analysis of the gender bias of a students’ grade predictor. The maximum gender bias of the network monotonically decreases with increasing precision.

between 4.1 and 4.6% among all quantizations, showing that the change in precision did not affect the quality of the network in a way that was perceivable by standard measures. However, our formal analysis confirmed a gender bias in the network, producing twins with a 15 to 21 difference in predicted math score. Surprisingly, the bias monotonically increased as the precision level in quantization lowered, indicating to us that quantization plays a role in determining the bias.

6 Conclusion

We introduced the first complete method for the verification of quantized neural networks which, by SMT solving over bit-vectors, accounts for their bit-precise semantics. We demonstrated, both theoretically and experimentally, that bit-precise reasoning is necessary to accurately ensure the robustness to adversarial attacks of a quantized network. We showed that robustness and non-robustness are non-monotonic in the number of bits for the numerical representation and that, consequently, the analysis of high-bits or real-numbered networks may derive false conclusions about their lower-bits quantizations. Experimentally, we confirmed that real-valued solvers produce many spurious results, especially on low-bit quantizations, and that also gradient descent may miss attacks. Additionally, we showed that quantization indeed affects not only robustness, but also other properties of neural networks, such as fairness. We also demonstrated that, using our balanced encoding, off-the-shelf SMT-solving can analyze networks with hundreds of neurons which, despite hitting the limits of current solvers, establishes an encouraging baseline for future research.

Acknowledgments

An early version of this paper was put into the easychair repository as EasyChair Preprint no. 1000. This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23(RiSE/SHiNE) and Z211-N23 (Wittgenstein Award), in part by the Aerospace Technology Institute (ATI), the Department for Business, Energy & Industrial Strategy (BEIS), and Innovate UK under the HICLASS project (113213).

References

1. Students performance in exams. <https://www.kaggle.com/spscientist/students-performance-in-exams>
2. Barocas, S., Hardt, M., Narayanan, A.: Fairness in machine learning. In: *Proceeding of NIPS (2017)*
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: *International Conference on Computer Aided Verification*. pp. 171–177. Springer (2011)
4. Bunel, R.R., Turkaslan, I., Torr, P.H.S., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: *NeurIPS*. pp. 4795–4804 (2018)
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer (2008)
6. Dutertre, B.: Yices 2.2. In: *International Conference on Computer Aided Verification*. pp. 737–744. Springer (2014)
7. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: *NFM. Lecture Notes in Computer Science*, vol. 10811, pp. 121–138. Springer (2018)
8. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: *ATVA. Lecture Notes in Computer Science*, vol. 10482, pp. 269–286. Springer (2017)
9. Evtimov, I., Eykholt, K., Fernandes, E., Kohno, T., Li, B., Prakash, A., Rahmati, A., Song, D.: Robust physical-world attacks on deep learning models. *arXiv preprint arXiv:1707.08945* 1 (2017)
10. Gehr, T., Mirman, M., Drachler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: *IEEE Symposium on Security and Privacy*. pp. 3–18. IEEE (2018)
11. Hadarean, L., Hyvarinen, A., Niemetz, A., Reger, G.: Smt-comp 2019. <https://smt-comp.github.io/2019/results> (2019)
12. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: *CAV (1). Lecture Notes in Computer Science*, vol. 10426, pp. 3–29. Springer (2017)
13. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A.G., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: *CVPR*. pp. 2704–2713. IEEE Computer Society (2018)

14. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: CAV (1). Lecture Notes in Computer Science, vol. 10426, pp. 97–117. Springer (2017)
15. Krizhevsky, A., Hinton, G.: Convolutional deep belief networks on cifar-10. Unpublished manuscript **40**(7) (2010)
16. Moosavi-Dezfooli, S., Fawzi, A., Frossard, P.: Deepfool: A simple and accurate method to fool deep neural networks. In: CVPR. pp. 2574–2582. IEEE Computer Society (2016)
17. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: ICML. pp. 807–814. Omnipress (2010)
18. Narodytska, N., Kasiviswanathan, S.P., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: AAAI. pp. 6615–6624. AAAI Press (2018)
19. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. JSAT **9**, 53–58 (2014)
20. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 243–257. Springer (2010)
21. Pulina, L., Tacchella, A.: Challenging SMT solvers to verify neural networks. AI Commun. **25**(2), 117–135 (2012)
22. Schönherr, L., Kohls, K., Zeiler, S., Holz, T., Kolossa, D.: Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding. In: accepted for Publication, NDSS (2019)
23. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. In: POPL. ACM (2019)
24. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. CoRR **abs/1312.6199** (2013)
25. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming (2018)
26. Tsipras, D., Santurkar, S., Engstrom, L., Turner, A., Madry, A.: Robustness may be at odds with accuracy. In: International Conference on Learning Representations (2019)
27. Xiang, W., Tran, H., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learning Syst. **29**(11), 5777–5783 (2018)
28. Zhao, Y., Shumailov, I., Mullins, R., Anderson, R.: To compress or not to compress: Understanding the interactions between adversarial attacks and neural network compression. In: SysML Conference (2019)

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

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





Highly Automated Formal Proofs over Memory Usage of Assembly Code

Freek Verbeek^{1,2}, Joshua A. Bockenek¹, and Binoy Ravindran¹

¹ Virginia Tech, Blacksburg VA, USA

² Open University of The Netherlands, Heerlen, The Netherlands



Abstract. We present a methodology for generating a characterization of the memory used by an assembly program, as well as a formal proof that the assembly is bounded to the generated memory regions. A formal proof of memory usage is required for compositional reasoning over assembly programs. Moreover, it can be used to prove low-level security properties, such as integrity of the return address of a function. Our verification method is based on interactive theorem proving, but provides automation by generating pre- and postconditions, invariants, control-flow, and assumptions on memory layout. As a case study, three binaries of the Xen hypervisor are disassembled. These binaries are the result of a complex build-chain compiling production code, and contain various complex and nested loops, large and compound data structures, and functions with over 100 basic blocks. The methodology has been successfully applied to 251 functions, covering 12,252 assembly instructions.

Keywords: Formal Verification · Assembly · x86-64 · Memory Usage

1 Introduction

This paper presents a formal methodology for reasoning over the *memory usage* of functions in a software suite. Various security properties require knowledge on memory usage. For example, proving absence of buffer overflows requires proving that a function does not write outside certain memory regions. Control-flow integrity requires showing, among other things, that the return address cannot be overwritten [61]. The security property called non-interference requires reasoning over which parts of the memory are used by which functions [50].

Moreover, memory usage is crucial for *compositional* reasoning over assembly code. Typically, compositional reasoning requires proving that certain code fragments are spatially independent [45,47]. A proof of memory usage can be used to prove such independence, thereby allowing composition. Consider a function g that at some point calls function f . Compositional reasoning means that a verification effort over f can be reused for verification of g without unfolding it. This *at least* requires that the verification effort over f establishes that f does not modify the stack frame of g . More generally, compositional reasoning requires

at least knowing that f restricts itself to certain parts of the memory. This is exactly what is established by proving memory usage.

Memory usage cannot satisfactorily be expressed at the source-code level. As an illustration, consider formulating a property that a function cannot overwrite its own return address. This requires knowledge on the values of the stack and frame pointers, making it an *assembly-level* property. At the assembly level, one can easily express a property formulating that the memory at the top of the stack frame (where the return address is stored) should remain unmodified.

Reasoning over assembly, however, is complicated due to the semantical gap between assembly and source code. In assembly code, ostensibly simple computations can be implemented using complex sequences of low-level operations. For example, a simple integer division by 10 can be implemented with a series of bit-level operations. Assembly code does not have types. It is common to, e.g., mix logical bitwise operators with signed integer arithmetic, or floating-point operations with bitvector operations. Assembly code does not have a clear distinction between stack frame and heap. Whether some address refers to a local variable stored in the stack, a global variable, or part of the heap, is provable only by adding assumptions on memory layout. Finally, assembly does not have a clear notion of scoping. Function calls are not necessarily clearly delineated, and instead of assuming that a function cannot write to a variable it has no access to (such as a local variable of another function), this has to be proven.

The contribution of this paper consists of a formal, compositional and highly automated methodology for reasoning over memory usage at the assembly-level.³ Our approach first uses untrusted tools to generate a *formal memory usage certificate* (see Section 2). This certificate contains 1.) theorems on memory usage, 2.) the preconditions under which memory usage can be shown, and 3.) *proof ingredients*. These proof ingredients contain assumptions on memory layout, control-flow information, and invariants. Section 2 provides an example of a function that theoretically can overwrite its own return address. We show that the certificate provides preconditions and a formal proof that a return-address-based exploit is not possible under those preconditions.

The certificate and the original assembly are loaded into an interactive theorem prover (ITP). Memory usage in general is an undecidable property (Rice’s theorem [48]), which is why we aim for an ITP environment to allow user interaction when necessary. Using the proof ingredients, the certificate is formally proven correct with minimal user interaction, making use of customized proof strategies. Section 3 describes certificate verification and composition.

To demonstrate applicability and scalability, we apply the methodology to x86-64 binaries of the Xen hypervisor [13] (see Section 4). The binaries are obtained via the standard Xen build process, including optimizations. The binaries are decompiled to assembly using off-the-shelf disassembly tools. Our methodology is applied to 251 functions; for each function a certificate is automatically generated, and a proof is finished in the Isabelle/HOL theorem prover [44]. With-

³ All code and proofs are publicly available [57].

out exception, the manual interaction consists of elementary interactive theorem proving such as applying the proper proof method.

While past work [38,41,25] on assembly-level formal verification exists, the degree of either scalability or automation is limited. As example of interactive theorem proving, Boyer and Yu verified machine-code implementations of various standard sort- and string functions, requiring over 19,000 lines of manually written proof code for the verification of roughly 900 instructions [8]. As example of automated theorem proving, Tan et al. presented an approach which takes about 6 hours for a 533-instruction string search algorithm [56]. In contrast, this paper involves a degree of user interaction of ≈ 85 lines of proof code per 1,000 lines of assembly. Our work is able to almost fully automatically verify 12,252 instructions from real world industrial binaries compiled by a real world build process. Section 5 discusses prior art, its contrast with the paper’s work, and the paper’s contributions. To the best of our knowledge, there is no related work that is able to achieve similar scalability and automation on real world binaries.

2 Formal Memory Usage Certificates

Figure 1 provides an example of a formal memory usage certificate (FMUC). The FMUC is generated automatically from an assembly file. This assembly file may be produced from a binary using a disassembler such as `objdump`, `IDA`,⁴ Ghidra’s decompiler,⁵ or Capstone [46]. In case source code is available, the assembly code can also be produced directly by a compiler. In this example, the C code of Figure 1a is used solely for presentation, the input to the FMUC generation is the assembly created by decompiling the corresponding binary. For each function in the assembly file, an FMUC is produced. External functions, for example due to dynamic linking, are treated as black boxes (see Section 3.4).

An FMUC consists of two parts: a *memory usage theorem* and its proof (see Figure 1c). The theorem consists of assumptions implying a Hoare triple [28,40] over the function. The Hoare triple is specific to memory usage. Intuitively, it means that from a state satisfying precondition P , after execution of code fragment f , the state satisfies postcondition Q (as in normal Hoare triples). The Hoare triple also contains a *memory region set* M . Besides its regular meaning, the Hoare triple expresses that any write that occurs during execution of f occurs within one of the memory regions in this set.

The term memory usage formally denotes an overapproximation of the memory written to by a function. Thus, any address that is not enclosed in one of the regions of M , is guaranteed to be preserved. Set M , however, will also include the memory regions read by the function, for verification purposes.

The precondition P expresses that the instruction pointer `rip` is at the entry point of the function. It also provides initial symbolic values for all registers and memory regions that are read (e.g.,: `rsp = rsp0`). Finally, it formulates that the return address is stored at the top of the stack frame. The postcondition Q

⁴ <https://www.hex-rays.com/products/ida/index.shtml>

⁵ <https://ghidra-sre.org/>

<pre> int main(int argc, char* argv[]) { int* a = (int*)argv; int* b = (int*)(argv + 4); *(int*)(argv + 2) = *a + *b; *(char*)argv = 'a'; int array[argc]; for (int i = 0; i < argc; i++) { array[i] = argv[i][0] * 2; } if (is_even(argc)) { return array[argc]; } return array[0]; } </pre> <p style="text-align: center;">(a) C Code</p>	<pre> Block 1149->120b; Loop Block 123e->1244; If SF ≠ OF Then Block 120d->123a Else Break Fi Pool; Block 1246->1249; Block 124b->124b; - call to is_even Block 1250->1252; If ZF Then Block 1263->1267 Else Block 1254->1261 Fi; Block 1269->1279; If ZF Then Block 1280->1285 Else Block 127b->127b Fi </pre> <p style="text-align: center;">(b) Syntactic Control Flow <i>f</i></p>
---	---

thm: $MRR \implies \{P\}f\{Q; M\}$

proof:

```

apply (check_scf_step)+
apply (check_scf_while "P123e || P1246")
apply (check_scf_step)+
                
```

where:

```

P ≡ rip = 1149 ∧ rsp = rsp0 ∧ ... ∧ *[rsp, 8] = ret_addr
Q ≡ rip = ret_addr ∧ rsp = rsp0 + 8 ∧ ... ∧ *[rsp, 8] = ret_addr
                
```

(c) Theorem and proof code

$M = \{a = [\text{rsp}_0, 8], b = [\text{fs}_0 + 40, 8], c = [\text{rsi}_0 + 36, 4], d = [\text{rsp}_0 - 8, 8], \dots\}$
 $MRR = \{a, b, c, d, \dots\}$ are separate

(d) The memory regions and their relations for block 123e->1244.

```

P123e(σ) = rip      = 123e
          rbp      = rsp0 - 8
          rdi      = rdi0
          rsp      = rsp0 - (88 + 16 * ((15 + 4 * sextend((31, 0)rdi0) / 16))
          *[rsp0 - 40, 8] = rsp0 - (85 + 16 * ((15 + 4 * sextend((31, 0)rdi0) / 16)) >> 2 << 2
          *[rsp0 - 48, 8] = sextend((31, 0)rdi0) - 1
          *[rsp0 - 56, 8] = rsi0 + 32
          ...
                
```

(e) Invariant at line 0x123e (only 7 out of 23 equations shown)

$$\{P_{124b}\} \boxed{\text{is_even}} \{P_{1250}; M_{\text{is_even}}\}$$

(f) Assumption due to call of function is_even

Fig. 1: An FMUC. Region $[a, s]$ denotes a region of s bytes starting at 64-bit address a . Notation $*r$ denotes reading region r in little-endian fashion. Notation $\langle 31, 0 \rangle \text{rdi}_0$ takes the lower 32 bits of the register.

expresses that the function has returned, i.e., the instruction pointer is equal to the return address and the stack pointer `rsp` is equal to its original value plus eight. For any callee-saved register, i.e., any register whose value is assumed to be preserved by the function call, it will say that its value is unchanged.

The component f of the memory usage theorem is a representation of the control flow of the function in terms of syntactic structures such as basic blocks, loops and if-then-else statements (see Figure 1b). We call this the syntactic control flow (SCF). The SCF is automatically generated from the control flow graph (CFG). The reason that a syntactic structure is required, is because the proof is done using Hoare logic, which is guided by syntax. The proof of an FMUC of an entire function is based on FMUCs per basic block. Thus one FMUC is generated per basic block, and one corollary FMUC for the entire function.

The proof consists of two further proof ingredients: *memory region relations* and *invariants*. We zoom in on block `123e`→`1244` to explain both of these. The FMUC provides 13 regions for this block, of which 4 are shown (see Figure 1d). Region *a* stores the return address. Region *b* depends on the segment register `fs` and stores the canary [15]. Region *c* is based on the pointer passed as second argument to the function. Finally, region *d* is part of the stack frame. The generated memory region relations assume that all these regions are separate. Out of the per-block memory regions and their relations, memory regions and relations for the function as a whole are composed.

For each basic block, an *invariant* is generated. Stronger invariants can lead to a tighter approximation of memory usage. The invariant assigned to block `123e`→`1244` is effectively a loop invariant (see Figure 1e). The frame pointer `rbp` is equal to the original stack pointer minus eight. Register `rdi` has not been touched. We also show some of the more complex invariants, such as the value of the stack pointer. In total, the loop invariant provides information on 11 registers and 12 memory locations for this basic block. Note that the FMUC provides preconditions in terms of the initial state of the corresponding basic block. In Section 3.2 these are lifted to preconditions in terms of the initial state of the function.

For this example, we treated `is_even` as an external function (see Figure 1f). An assumption was thus generated, that expresses that the memory usage of that function suffices to show that the invariant at line `124b` implies the invariant at line `1250`. This means, among others, that the memory used by `is_even` (denoted M_{is_even}) should not overlap with regions *a* through *d*. Section 3.4 provides more information on composition.

The FMUC is generated automatically, except for the three line proof in Figure 1c. Due to the undecidability of memory usage, interaction may be required. Isabelle/HOL proof strategies are provided to assist in that interaction. Section 3 provides more details. The manual effort required in proving the FMUC for this function, consists simply of calling the proper proof strategies. First, `check_scf_step` is run, applying Hoare logic rules and proving correctness of the memory usage until the loop. Then, the proof strategy for dealing with the loop

is called, with the invariant generated from the FMUC. Finally, `check_scf_step` is called again, which is able to verify the remainder of the function.

Finally, note that without any assumptions the function could overwrite its own return address at various places. The memory region relations MRR are sufficiently strong to exclude this. These relations thus form the preconditions under which a return-address exploit is impossible. As example, they assume that regions a and c are separate. This means that the address stored in parameter `argv` (reflected as `rsi0` at the assembly level) is not allowed to point to a region within the stack frame of function `main`.

Due to space restriction, we omit details on the algorithms that generate an FMUC. In general, none of the FMUC generation is part of the trusted computing base. That is, none of the algorithms need to be backed up by formal proofs. The *output* of the FMUC generation is imported into Isabelle/HOL, where it is proven correct. If there is an error in CFG generation, control flow extraction, symbolic execution, or in the generated invariants, then the certificate cannot be proven in Isabelle/HOL. One exception is the memory region relations. They are assumptions, and if they are internally inconsistent this leads to a vacuous truth. For that reason, Z3 is used to generate them [39], making it impossible to introduce, e.g., a relation where two overlapping regions are considered separate.

3 FMUC Verification

This section presents the verification of an FMUC. Both the FMUC and the original assembly are loaded into Isabelle/HOL. The theorem is then proven using the proof ingredients stored in the FMUC. This means that given a step function that models the semantics of the assembly instructions, the Hoare triple is verified.

Let $\text{step} :: I \times S \times S \mapsto \mathbb{B}$ be a transition relation. It takes as input an instruction of type I and two states σ and σ' . It returns true if and only if execution of the instruction in state σ can produce state σ' . Undefined behavior, such as null-pointer dereferencing, is modeled by relating a state to any successor state. The semantics of a syntactic control flow (SCF) are straightforwardly defined by a function $\text{exec_scf} :: SCF \times S \times S \mapsto \mathbb{B}$ (here SCF denotes the type of a syntactic control flow object). In case of loops the function is defined using a least fixed point construction. This way, if the halting condition is never met, there exists no related σ' .

First, we define the notion of memory usage wrt. a certain state change:

Definition 1. *The set of memory regions M is the memory usage wrt. the state change from σ to σ' , if and only if, any byte at an address a not inside one of the regions is unchanged.*

$$\text{usage}(M, \sigma, \sigma') \equiv \forall a \cdot (\forall r \in M \cdot [a, 1] \bowtie r) \implies \sigma' : *[a, 1] = \sigma : *[a, 1]$$

Here, notation $\sigma : *[a, s]$ means reading in little-endian fashion s bytes from memory address a in state σ . Notation $r_0 \bowtie r_1$ denotes that two regions are separate.

Definition 2. *A memory usage Hoare triple is defined as:*

$$\{P\} f \{Q; M\} \equiv \forall \sigma \sigma' \cdot P(\sigma) \wedge \text{exec_scf}(f, \sigma, \sigma') \longrightarrow Q(\sigma') \wedge \text{usage}(M, \sigma, \sigma')$$

In words, Definition 2 states the following: if precondition P holds on the initial state σ and σ' can be obtained by executing f , postcondition Q holds on the produced state and the values stored in all memory regions outside set M are preserved.

3.1 Verification Tools Used

Isabelle/HOL The theorem prover utilized in this work was Isabelle 2018 [44]. It is a generic tool with a flexible, extensible syntactic framework. Isabelle also utilizes a powerful proof language known as intelligible semi-automated reasoning (Isar) [59] and a proof strategy language called Eisbach [37]. We made heavy use of Word library [17]. This library provides a limited-precision integer type, `'a word`, where `'a` is the number of bits in the integer. Various operations are provided for manipulation of and arithmetic involving formal words, including bit indexing, bit shifting, setting specific bits, and signed and unsigned arithmetic. Operators for inequality are also included, as well as operations for converting between word sizes.

Machine Model and Instruction Semantics Heule et al. provide semantics of the x86-64 architecture [27]. Instead of manually codifying instruction semantics, they applied machine learning to derive semantics from a live x86 machine. This produced highly reliable semantics: they compared the semantics to manually written semantics based on the Intel reference manuals, and found that in the few cases where they differed the Intel manuals were wrong. Roessle et al. embedded these semantics into the Isabelle/HOL theorem prover and tested the formal Isabelle semantics against live x86 hardware [49]. This formal machine model is the base of our verification effort.

Symbolic Execution Bockenek et al. provide an Isabelle/HOL symbolic execution engine based on the above semantics [6]. Effectively, this provides a function `symb_exec` that symbolically runs basic blocks. Let a_0 and a_1 be the start- and end-addresses of the block. A call to `symb_exec($a_0, a_1, \sigma, \sigma'$)` returns true if and only if state σ' is the result of symbolically executing the block from state σ . The symbolic execution is completely written in Isabelle/HOL, meaning that every rewrite rule has been formally proven correct.

3.2 Per-block Verification

Verification occurs by first verifying per basic block. Figure 2a shows an introduction rule for establishing a Hoare triple over a basic block. The first assumption requires the symbolic execution method to run over a universally quantified symbolic state σ that satisfies the precondition. Any resulting state σ' should satisfy the postcondition Q , and the set of memory regions M generated for the block should be correct.

The second assumption is required because of an important subtlety: the regions generated in the FMUC are expressed in terms of the initial state of their basic block. However, it makes no sense to express the regions used by individual blocks within a larger function in terms of their own initial state. If a region of a basic block somewhere within a function body depends on, e.g., the value of register `rdi` at the start of that block, then it is unsound to express that memory region in terms of `rdi0`, i.e., the value of `rdi` at the start of the function. Therefore, the Hoare triples are defined based on a set of memory regions M' that solely depends on the initial state of the function. For each block, that set is obtained by taking the generated set of memory regions M (expressed in terms of the initial state of the block) and applying it to any state that satisfies the current invariant. This produces a set of regions expressed in terms of the initial state of the function.

An Isabelle proof strategy has been implemented that, given the proof ingredients from the FMUC, discharges this introduction rule. The proof strategy runs symbolic execution within Isabelle/HOL, proves the postcondition and proves the memory usage. The open variables P , Q , a_0 , a_1 and M are all provided by the FMUC. No interaction is required; for basic blocks the proof is automated.

3.3 Verification of Function Body

$$\frac{\forall \sigma \sigma' \cdot P(\sigma) \wedge \text{symb_exec}(a_0, a_1, \sigma, \sigma') \implies Q(\sigma') \wedge \text{usage}(M(\sigma), \sigma, \sigma')}{M' = \{ r \mid \exists \sigma \cdot P(\sigma) \wedge r \in M(\sigma) \}}$$

$$\frac{}{\{P\} \text{Block } a_0 \rightarrow a_1 \{Q; M'\}}$$

(a) Introduction rule

$$\frac{\{P\} f \{Q; M_1\} \quad \{Q\} g \{R; M_2\} \quad M = M_1 \cup M_2}{\{P\} f; g \{R; M\}}$$

(b) Sequence rule

$$\frac{\{I \wedge B\} f \{I'; M\} \quad I' \implies I \quad I \wedge \neg B \implies Q}{\{I\} \text{While } B \text{ DO } f \text{ OD } \{Q; M\}}$$

(c) While rule

Fig. 2: Hoare rules for memory usage

For each syntactic construct, a Hoare rule is defined (see Figure 2). The sequence and conditional rules (only first is shown) are straightforward: the memory usage is the union of the memory usage of the constituents. Note that the sequence rule is sound only because the memory predicates are independent of the initial state of the basic blocks, as discussed above.

The while rule is based on a loop invariant I . If the memory usage of one iteration of function body f is constrained to the set of memory regions M , then

that holds for the entire loop. This sounds counterintuitive. Consider a simple C-like loop iterating from $i = 0$ while $i < 10$ and as body the assignment $a[i] = 0$, i.e., it writes to the i th element of an array. Verification of the loop requires the invariant $I(\sigma) = i(\sigma) < 10$. The FMUC of the loop body will have a set of memory regions $M(\sigma) = \{[a + i(\sigma), 1]\}$, i.e., one region of one byte, expressed in terms of the initial state of the basic block. Now consider the application of the introduction rule to the block of the loop body. It will introduce a Hoare triple with:

$$\begin{aligned} M' &= \{ r \mid \exists \sigma \cdot I(\sigma) \wedge r \in M(\sigma) \} \\ &= \{ r \mid \exists \sigma \cdot i(\sigma) < 10 \wedge r = [a + i(\sigma), 1] \} \\ &= \{ [a', 1] \mid a \leq a' \leq a + 10 \} \end{aligned}$$

The set M' is actually the memory used by the entire loop. This is because the introduction rule applies the state-dependent set of memory regions to any state that satisfies the invariant. This shows that the strength of the generated invariants influences the tightness of the overapproximation of memory usage. A weaker invariant, e.g., $i < 20$, would produce a larger set of memory regions.

An Isabelle/HOL proof strategy is implemented that automatically applies the proper Hoare logic rule. It is driven by the syntactic control flow provided by the FMUC. For function bodies without loops, this proof strategy requires no further interaction. For each loop entry, it is required to manually apply the weaken rule to show that the postcondition of the block before entry implies the loop invariant. Without exception, each of these proofs could be finished using standard off-the-shelf Isabelle/HOL tools. The part that is usually the most involved – defining the invariants – is taken care of by the FMUC generation.

3.4 Composition

Let f be a function body. Assume that the function has been verified, i.e., a Hoare triple has been proven of the form: $\{P_f\} f \{Q_f; M_f\}$. In order to composably reuse that verification effort, function f is considered to be a black box once it is verified. Now consider a function g calling function f :

```
a0: push rbp
a1: call f
a2: pop rbp
a3: ret
```

Let P denote the precondition right before executing the assembly instruction `call`. Precondition P contains the equality $*[\mathbf{rsp}_0^g - 8, 8] = \mathbf{rbp}_0^g$, expressing that function g has pushed frame pointer `rbp` into its own local stack frame. Let Q denote the postcondition just after returning, but before executing `pop`. The postcondition of g expresses that callee-saved register `rbp` is properly restored, i.e., $\mathbf{rbp} = \mathbf{rbp}_0^g$. That is indeed done by the `pop` instruction. In order to prove proper restoration of `rbp`, it must be proven that function f did not overwrite any byte in region $[\mathbf{rsp}_0^g - 8, 8]$. Additionally, function f must be proven not to overwrite region $[\mathbf{rsp}_0^g, 8]$ which stores the return address of g . For this particular instance of calling f , it thus must be proven that f preserves these two regions.

More generically, function f can be called by various functions other than g . For each call the specific requirements on which memory regions are required to be preserved differ. Thus, to be able to verify function f once, and reuse that verification effort for each call, the verification effort must at least contain an overapproximation of the memory written to by function f . Note that this is exactly the requirement when using *separation logic* [45,47,33]. Separation logic provides a *frame rule* for compositional reasoning. This frame rule informally states that if a program can be confined to a certain part of a state, properties of this program carry over when the program is part of a bigger system.

We thus provide a version of the frame rule of separation logic, specific to memory usage verification (see Figure 3). Effectively, this rule is used to prove that the memory usage of a caller function g is equal to the memory it uses itself, *plus* the memory used by function f . It requires four assumptions. First, it assumes function f has been verified for memory usage, with M_f denoting that memory usage. Second, it assumes that precondition P can be split up into two parts: precondition P_f required to verify function f , and a separate part P_{sep} . The separate part is specific to the actual call of the function. In the example, P_{sep} will contain the equality $[\text{rsp}_0^g - 8, 8] = \text{rbp}_0^g$. Third, the correctness of the set of memory regions M_f should suffice to prove that the separated part P_{sep} is preserved. In the example, this effectively means that M_f should not overlap with the two regions of g . Fourth, P_{sep} and Q_f should imply postcondition Q .

$$\begin{array}{c}
 \{P_f\} f \{Q_f; M_f\} \\
 P \implies P_f \wedge P_{\text{sep}} \\
 \forall \sigma \sigma' \cdot \text{usage}(M_f, \sigma, \sigma') \wedge P_{\text{sep}}(\sigma) \longrightarrow P_{\text{sep}}(\sigma') \\
 Q_f \wedge P_{\text{sep}} \implies Q \\
 \hline
 \{P\} \text{Call } f \{Q; M_f\}
 \end{array}$$

Fig. 3: Frame rule for composition of memory usage

In practice, many functions will not be part of the assembly code under verification (e.g., external calls). We thus have to generate the assumptions required to proceed with verification. To this end, we introduce the following notation:

$$\{P\} \boxed{\text{f}} \{Q; M_f\} \equiv \exists P_f Q_f P_{\text{sep}} \cdot \text{four assumptions of frame rule are satisfied}$$

Making this assumption informally expresses that function f is assumed to have been verified. Its memory usage M_f is assumed to suffice to prove that we could step from states satisfying P to states satisfying Q .

4 Case Study: Xen Project

The Xen Project [13] is a mature, widely-used virtual machine monitor (VMM), also known as a *hypervisor*. Hypervisors provide a method of managing multiple

virtual instances of operating systems (called guests or domains) on a physical host. The Xen hypervisor is a suitable case study because of its security relevance and its complex build process involving real production code. Security is a significant issue in environments where hypervisors are used, such as the Amazon Elastic Compute Cloud (Amazon EC2), Rackspace Cloud, and many other cloud service providers. For example, when one or more physical hosts support virtual guests for any number of distinct users, ensuring isolation of the guest operating systems (OSs) is important. The Xen build process produces multiple binaries that contain functions not present in the Xen source itself. This is due to the inclusion of external static libraries and programs. We used Xen 4.12 compiled with GCC 8.2 via the standard Xen build process. This build process uses various optimization levels, ranging from 01 to 03.

Of the binaries produced by the Xen build process, we considered three: `xenstore`, `xen-cpuid`, and `qemu-img-xen`. The `xenstore` binary is involved in the functionality of XenStore,⁶ a hierarchical data structure shared amongst all Xen domains. The `xen-cpuid` utility queries the underlying processors and displays information about the features they support. The third binary, `qemu-img-xen`, consists of over three hundred functions that are not present in the Xen source code. It provides some of the functionality of Quick Emulator (QEMU). QEMU is a free, open-source emulator.⁷ Xen uses it to emulate device models (DMs), which provide an interface for hardware storage.

Binaries	Function Count	Instruction Count	Loops	Manual Lines of Proof
<code>xenstore</code>	2/6	100	0	6
<code>xen-cpuid</code>	2/3	210	2	39
<code>qemu-img-xen</code>	247/343	11,942	64	1,002
Total	251/352	12,252	65	1,047

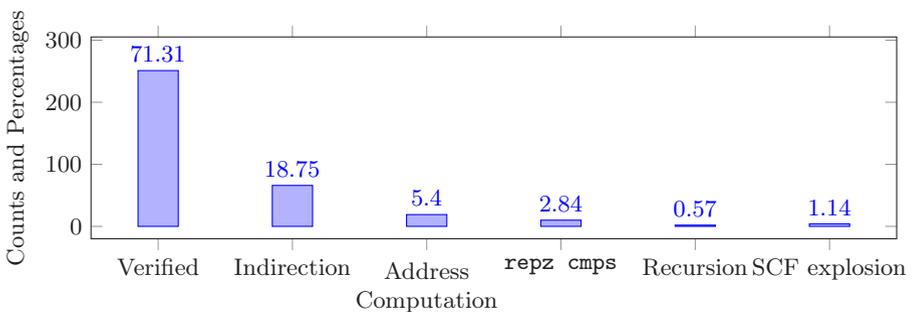


Fig. 4: Case Study Overview

⁶ <https://wiki.xen.org/wiki/XenStore>

⁷ <https://www.qemu.org/>

Our methodology is currently capable of dealing with 71% of the functions present in these binaries (see Figure 4). The supported features include (nested) loops, subcalls, variable argument lists, jumps into other function bodies, string instructions with the `rep` prefix. There is no particular limit on function size. The average number of instructions per function analyzed is 49. Some of the functions analyzed have over 300 instructions and over 100 basic blocks.

There are five categories of features we do not support. The first and most common is *indirection*, accounting for 19%. Indirection involves a call or jump instruction that loads the target address from a register or memory location rather than using a static value. Switch statements and certain uses of `goto` are the most common causes of indirect jumps. Indirect calls generally result from usage of function pointers. For example, the `main` functions of all three verified binaries used switch statements in loops in the process of parsing command line options. These statements introduced indirect branches.

The second category involves issues related to generating the memory region relations. This step requires solving linear arithmetic over symbolically computed addresses. Sometimes, addresses are computed using a combination of arithmetic operators with bitwise logical operators. In some of these cases, our translation to Z3 does not produce an answer. As an example, function `qcow_open` uses the rotate-left function to compute an address. As another example, function `AES_set_encrypt_key` produces addresses that are obtained via combinations of bit-shifting, bit masking, and xor-ing.

The instruction `repz cmps` is currently not supported for technical reasons. It is the assembly equivalent of the function `strncmp`, but instead writes its result to a flag. Various other string-related instructions with the `rep` prefix are supported. Functions with *recursion*, a minority in systems code, are also not supported. Recursive stack frames in our framework are not well-suited to automation. The two recursive functions we encountered both perform file-system-like tasks. Functions `do_chmod` and `do_ls` are similar respectively to the permission-setting `chmod` utility, and directory-displaying `ls`. The final category is functions whose SCF explodes. The issue occurs mostly when loops have multiple entries.

The table in Figure 4 provides an overview of the verification effort. The table shows the absolute counts of functions verified as well as the total number of instructions for those functions. Alongside that information is the number of functions with loops that were verified and how many manual lines of proof were required in total. The vast majority of those manual proof lines were related to the loop count.

5 Related Work

Assembly verification has been an active research field for decades. Table 1 provides an inexhaustive overview of related work. We first address some formal verification efforts at the assembly level. Then we discuss work in which assembly verification played a role in a larger verification context. Finally, verified compilation and static binary analysis tools are discussed.

Assembly-level Verification. Clutterbuck et al. [14] performed formal verification of assembly code using SPACE-8080, a verifiable subset of the Intel 8080 instruction set architecture (ISA) that is analyzable and formally verifiable [12]. Not long after, Bevier et al. presented a systems approach to software verification [5,7]. Their work laid out a methodology for verifying the correctness of all components necessary to execute a program correctly, including compiler, assembler and linker. The methodology was applied to a small OS kernel, Kit [4]. Similarly, Yu and Boyer [60,8] presented operational semantics and mechanized reasoning for approximately 80% of the instructions of the MC68020 microprocessor, over 85 instructions. Their approach utilized symbolic execution of operational semantics. These early efforts required significant interaction. For example, the approach of Yu and Boyer required over 19,000 lines of manually written proof to verify approximately 900 assembly instructions.

Matthews et al. targeted a simple machine model called TINY as well as Java virtual machine (JVM) bitcode using the M5 operational model [38]. Their approach utilizes symbolic execution of code annotated with manually written invariants. It also used verification condition generation to increase automation. This reduced the number of manually written invariants. Both of these assembly-style languages feature a stack for handling scratch variables rather than a register file as x86, ARM, and most other mainstream ISAs do.

Goel et al. presented an approach for modeling and verifying non-deterministic programs on the binary level [25,24]. In addition to formulating the semantics of most user-mode x86 instructions, they provided semantics for common system calls. System call semantics increase the spread of programs that can be fully verified. Their work was applied to multiple small case studies, including a word count program and two kernel-mode memory copying examples.

Bockenek et al. provide an approach to proving memory usage over x86 code [6]. They used a Floyd-style reasoning framework to prove Floyd invariants over functions [21]. They have applied it to functions of the HermitCore unikernel, covering 2,613 assembly instructions. Their approach required a significant amount of manual effort: pre- and postconditions, invariants, the actual regions of memory used and their relations all need to be manually defined.

The main difference between these existing approaches and the methodology presented in this paper concerns automation. Generally, interactive theorem proving over semantics of assembly instructions does not scale due to the amount of intricate user interaction involved. Figure 1e shows, e.g., the complexity of defining an assembly-level invariant even for a small example. Fully automated approaches to formal verification, however, do not scale either. The recent automated approach AUSPICE takes about 6 hours for a 533-instruction string search algorithm [56]. To the best of our knowledge, our methodology is the first that is able to deal with optimized x86-64 binaries produced by production code, with a “manual effort vs. instruction count ratio” of roughly 1 to 11.

Myreen et al. developed *decompilation-into-logic* [40,41,42]. That work, developed in the HOL4 theorem prover [54], uses operational semantics of machine code to lift programs into a functional form. That functional form can then be

Table 1: Overview of Related Work.

Work	Target	Approach	Applications	Verified code
Clutterbuck & Carré	SPACE-8080	ITP	N/A	
Bevier et al.	PDP-11-like	ITP	Kit	
Yu & Boyer	MC68020	ITP	String functions	863 insts
Matthews et al.	Tiny/JVM	ITP+VCG	CBC enc/dec	631 insts
Goel et al.	x86-64	ITP	word-count	186 insts
Bockenek et al.	x86-64	ITP	HermitCore	2,613 insts
Tan et al.	ARMv7	ATP	String search	983 insts
Myreen et al.	ARM/x86	DiL	seL4	9,500 SLoC
Feng et al.	MIPS-like	ITP	Example functions	
This paper	x86-64	ITP+CG	Xen	12,252 insts
Sewell et al.	C	TV+DiL	seL4	9,500 SLoC
Shi et al.	C/ARM9	ATP+MC	ORIENTAIS	8,000 SLoC, 60 insts
Dam et al.	ARMv7	ATP+UC	PROSPER	3,000 insts
VCG = Verification Condition Generation		DiL = Decompilation-into-Logic		
SLoC = Source Lines of Code		ATP = Automated Theorem Proving		
UC = User Contracts		CG = Certificate Generation		
TV = Translation Validation		MC = Model Checking		

used in a Hoare logic framework for program analysis [40]. Decompilation-into-logic has been used for both ARM and x86 ISA machine models, and applied to various large examples, including benchmarks such as a garbage collector, and the Skein hash function. Decompilation-into-logic covers – formally – the gap between machine code and a HOL model. It is not a verification method in itself, i.e., it does not verify properties over the machine code. It can be used as a component in a binary-level verification methodology [51].

Feng et al. presented stack abstractions for modular verification of assembly code [20,19]. Their work allows for integration of various proof-carrying code systems [43]. As with our work, it utilizes a Hoare-style framework for its verification. The authors applied their work to multiple example functions, such as two factorial implementations. In contrast to our approach, manual annotations are required to provide information regarding invariants and memory layout.

Integrated Assembly-Level Verification Efforts. A major verification effort, based on decompilation-into-logic, is the verification of the seL4 kernel [32,31]. The seL4 project provides a microkernel written in formally proven correct C code. The tool AutoCorres [26] is used for C code verification. Sewell et al. verified a refinement relation between the C source code and an ARM binary for both non-optimized and optimized at 02 [51]. The major differences with respect to our work is that our methodology targets existing production code, instead of code written with verification in mind. For example, the seL4 source code does not allow taking the addresses of stack variables (such as in Figure 1a): their approach requires a static separation of stack and heap. Neither the seL4 proof effort nor our methodology support function pointers.

Shi et al. formally verified a real-time operating system (RTOS) for automotive use called ORIENTAIS [52]. Part of their approach involved source-level verification using a combination of Hoare logic and abstract communicating sequential processes (CSP) model analysis [29]. Binary verification was done by lifting the RTOS binary to xBIL, a related hardware verification language [53]. They translated requirements from the OSEK automotive industry standard to source code annotations.

Targeting a similar case study as this paper, Dam et al. formally verified a tiny ARMv7 hypervisor, PROSPER [16,3] at the assembly level. Their methodology integrated HOL4 with the Binary Analysis Platform (BAP) [9]. BAP utilizes a custom intermediate language that provides an architecture-agnostic representation of machine instructions and their side effects. HOL4 was used to translate the ARM binary into BAP’s intermediate language, using the formal model of the ARM ISA by Fox et al. [22]. The SMT solver Simple Theorem Prover (STP) [23] was used to determine the targets of indirect branches and to discharge the generated verification conditions. While the approach was generally automated, user input was still required to describe software contracts of the hypervisor.

Verified Compilation. In contrast to directly verifying machine or assembly code, one can verify source code and then use *verified compilation*. Verified compilation establishes a refinement relation between assembly and source code. The CompCert project [36] provides a compiler for a subset of C. Its output has been verified to have the same semantics as the C source code. The seL4 project used CompCert to reduce its trusted code base [31]. Another example of verified compilation is CakeML [35]. It utilizes a subset of Standard ML modeled with big-step operational semantics. The main purpose of verified compilation, however, is not to verify properties over the code. For example, if the source code is vulnerable to a return-address exploit, then the assembly code is vulnerable as well. Verified compilation is thus often accompanied by source code verification. We have argued that for memory usage, assembly-level verification is necessary.

Static Analysis. Static analysis of binary code has been an active research field for decades [34,9,58]. The BitBlaze project [55] provides a tool called Vine which constructs control flow graphs for supplied programs and lifts x86 instructions to its own intermediate language (IL). Though Vine itself is not formally verified, it does support interfacing with the SMT solver STP as well as CVC [1,2]. The tool Infer [10], developed at Facebook, provides in-depth static analysis of LLVM code to detect bugs in C and C++ programs. It utilizes separation logic [47] and bi-abduction [11] to perform its analyses in an automated fashion. It is designed to be integrated into compiler toolchains, in order to provide immediate feedback even in continuous integration scenarios. FindBugs is a static analysis tool for Java code [30]. Rather than relying on formal methods, it uses searches for common code idioms to detect likely bugs. Common errors it highlights include null pointer dereferences, objects that compare equal not having equal hash codes, and inconsistent synchronization. The tool Splint [18] detects buffer overflows and similar potential security flaws in C code. It relies on annotated preconditions to derive postconditions.

The main difference between these static analysis tools and formal verification is that these tools generally are highly suited to find bugs, but are not able to prove absence of them. They generally apply techniques that are formally unsound, such as depth-bounded searches.

6 Conclusion

This paper presents an approach to formal verification of memory usage of functions in a compiled program. Memory usage is a property that expresses an overapproximation of the memory used by assembly code. Memory usage is fundamental to compositional verification of assembly code, as compositionality at least requires to prove that functions do not unexpectedly interfere with each others' stack frame. It can also be used to show security-related properties, such as integrity of the return address.

Our approach automatically generates a formal memory usage certificate that includes 1.) a set of memory regions read from and written to, 2.) postconditions that express sanity constraints over the function (e.g., the return address has not been overwritten, callee-saved registers are restored), 3.) proof ingredients such as the preconditions necessary for formal verification. The certificate is loaded into a theorem prover, where it is verified. Since the problem of memory usage is undecidable, we use an interactive theorem prover. The proof ingredients, combined with custom proof strategies, provide a large degree of automation. They deal with memory aliasing, the control flow of the function, and invariants.

The approach is applied to three binaries of the Xen hypervisor. These binaries contain production code and are the result of a complex build chain. They contain, among others, various nested loops, large and compound data structures, variadic functions, and both in- and external function calls. For 71% of the functions of these binaries, a certificate could be generated and verified. For each of these functions, it has at least been formally proven that the return address is not overwritten. The amount of user interaction is roughly 85 lines of proof code per 1,000 lines of assembly code. The greatest bottleneck is in indirect branching, which accounts for 19% of the functions.

In the near future we aim to support indirect branching. This would allow support of switches, callbacks, and pointers to functions. Additionally, we aim to strengthen the invariant generation. Stronger invariants lead to a tighter overapproximation of memory usage. The challenge here is not only to generate these invariants, but to automate their proof as well. Finally, we want to leverage the certificate to target high-level security properties, such as noninterference.

Data Availability Statement and Acknowledgments All code and proofs are available in the Zenodo repository: [10.5281/zenodo.3676687](https://doi.org/10.5281/zenodo.3676687). Distribution statement: Approved for public release; distribution is unlimited. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR.00112090028, ONR under grant N00014-17-1-2297, and NAVSEA/NEEC under grant N00174-16-C-0018.

References

1. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: International Conference on Computer Aided Verification. pp. 515–518. Springer (2004)
2. Barrett, C., Tinelli, C.: CVC3. In: International Conference on Computer Aided Verification. pp. 298–302. Springer (2007)
3. Baumann, C., Näslund, M., Gehrman, C., Schwarz, O., Thorsen, H.: A high assurance virtualization platform for armv8. In: 2016 European Conference on Networks and Communications (EuCNC). pp. 210–214. IEEE (2016)
4. Bevier, W.R.: Kit and the short stack. *Journal of Automated Reasoning* **5**(4), 519–530 (1989)
5. Bevier, W.R., Hunt, W.A., Moore, J.S., Young, W.D.: An approach to systems verification. *Journal of Automated Reasoning* **5**(4), 411–428 (Dec 1989). [10.1007/BF00243131](https://doi.org/10.1007/BF00243131)
6. Bockenek, J.A., Verbeek, F., Lammich, P., Ravindran, B.: Formal verification of memory preservation of x86-64 binaries (Sep 2019)
7. Boyer, R.S., Moore, J.S.: *A Computational Logic*. Academic Press, Inc. (1979)
8. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. *Journal of the ACM* **43**(1), 166–192 (1996)
9. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) International Conference on Computer Aided Verification. pp. 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). [10.1007/978-3-642-22110-1_37](https://doi.org/10.1007/978-3-642-22110-1_37)
10. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) *NASA Formal Methods*. pp. 459–465. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). [10.1007/978-3-642-20398-5_33](https://doi.org/10.1007/978-3-642-20398-5_33), <https://fbinfer.com/>
11. Calcagno, C., Distefano, D., O’Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. pp. 289–300. POPL ’09 (2009)
12. Carré, B.A., O’Neill, I.M., Clutterbuck, D.L., Debney, C.W.: SPADE—the southampton program analysis and development environment. In: *Software Engineering Environments*. Peter Peregrinus, Ltd. Stevenage (1986)
13. Chisnall, D.: *The Definitive Guide to the Xen Hypervisor*. Pearson Education (2008)
14. Clutterbuck, D.L., Carré, B.A.: The verification of low-level code. *Software Engineering Journal* **3**(3), 97–111 (May 1988). [10.1049/sej.1988.0012](https://doi.org/10.1049/sej.1988.0012)
15. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: *USENIX Security Symposium*. vol. 98, pp. 63–78. San Antonio, TX (1998)
16. Dam, M., Guanciale, R., Nemat, H.: Machine code verification of a tiny ARM hypervisor. In: *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*. pp. 3–12. TrustED ’13, ACM Press, New York, NY, USA (2013). [10.1145/2517300.2517302](https://doi.org/10.1145/2517300.2517302)
17. Dawson, J., Graunke, P., Huffman, B., Klein, G., Matthews, J.: Machine words in Isabelle/HOL (Aug 2018)

18. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. *IEEE Software* **19**(1), 42–51 (Jan 2002). [10.1109/52.976940](https://doi.org/10.1109/52.976940)
19. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. Tech. Rep. YALEU/DCS/TR-1336, Dept. of Computer Science, Yale University, New Haven, CT (Nov 2005), <http://flint.cs.yale.edu/publications/sbca.html>
20. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'06, vol. 41, pp. 401–414. ACM Press, New York, NY, USA (Jun 2006)
21. Floyd, R.W.: Assigning meanings to programs. *Mathematical Aspects of Computer Science* **19**(1), 19–32 (1967)
22. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) *Interactive Theorem Proving*. pp. 243–258. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). [10.1007/978-3-642-14052-5_18](https://doi.org/10.1007/978-3-642-14052-5_18)
23. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) *Computer Aided Verification*. pp. 519–531. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). [/10.1007/978-3-540-73368-3_52](https://doi.org/10.1007/978-3-540-73368-3_52)
24. Goel, S.: Formal Verification of Application and System Programs Based on a Validated x86 ISA Model. Ph.D. thesis (2016), <http://hdl.handle.net/2152/46437>
25. Goel, S., Hunt, W.A., Kaufmann, M., Ghosh, S.: Simulation and formal verification of x86 machine-code programs that make system calls. In: 2014 Formal Methods in Computer-Aided Design (FMCAD). pp. 91–98 (Oct 2014). [10.1109/FMCAD.2014.6987600](https://doi.org/10.1109/FMCAD.2014.6987600)
26. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Beringer, L., Felty, A. (eds.) *International Conference on Interactive Theorem Proving*. pp. 99–115. ITP 2012, Springer-Verlag, Berlin, Heidelberg (Aug 2012)
27. Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: Automatically learning the x86-64 instruction set. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 237–250. PLDI '16, ACM, New York, NY, USA (2016)
28. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM* **12**(10), 576–580 (Oct 1969)
29. Hoare, C.A.R.: Communicating sequential processes. *Commun. ACM* **21**(8), 666–677 (Aug 1978). [10.1145/359576.359585](https://doi.org/10.1145/359576.359585)
30. Hovemeyer, D., Pugh, W.: Finding bugs is easy. *SIGPLAN Not.* **39**(12), 92–106 (Dec 2004). [10.1145/1052883.1052895](https://doi.org/10.1145/1052883.1052895), <http://findbugs.sourceforge.net/>
31. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems* **32**(1), 2:1–2:70 (Feb 2014). [10.1145/2560537](https://doi.org/10.1145/2560537)
32. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220. ACM (2009)
33. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J.H., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: *European Symposium on Programming*. pp. 696–723. Springer (2017)

34. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: *USENIX Security Symposium*. vol. 14, pp. 11–11 (2005)
35. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 179–191. POPL '14, ACM, New York, NY, USA (2014), <https://cakeml.org/>
36. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - a formally verified optimizing compiler. In: *Embedded Real Time Software and Systems, 8th European Congress. ERTS 2016, SEE, HAL, Toulouse, France (Jan 2016)*, <https://hal.inria.fr/hal-01238879>
37. Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning* **56**(3), 261–282 (2016)
38. Matthews, J., Moore, J.S., Ray, S., Vroon, D.: Verification condition generation via theorem proving. In: *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. pp. 362–376. Springer-Verlag (2006)
39. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer-Verlag (2008)
40. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 568–582. Springer-Verlag, Berlin, Heidelberg (2007)
41. Myreen, M.O., Gordon, M.J.C., Slind, K.: Machine-code verification for multiple architectures - an application of decompilation into logic. In: *2008 Formal Methods in Computer-Aided Design*. pp. 1–8. IEEE (Nov 2008)
42. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic—improved. In: *2012 Formal Methods in Computer-Aided Design (FMCAD)*. pp. 78–81. IEEE (2012)
43. Necula, G.C.: Proof-carrying code. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. pp. 106–119. ACM (1997)
44. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283. Springer Science & Business Media (2002)
45. O’Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: *International Workshop on Computer Science Logic*. pp. 1–19. Springer (2001)
46. Quynh, N.A.: Capstone: Next-gen disassembly framework (Aug 2014), <http://www.capstone-engine.org/>, accessed June 27, 2019
47. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. pp. 55–74. IEEE (2002)
48. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society* **74**(2), 358–366 (1953)
49. Roessle, I., Verbeek, F., Ravindran, B.: Formally verified big step semantics out of x86-64 binaries. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. pp. 181–195. CPP 2019, ACM, New York, NY, USA (2019)
50. Rushby, J.: Noninterference, Transitivity, and Channel-Control Security Policies. SRI International, Computer Science Laboratory (1992)

51. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified OS kernel. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 471–482. PLDI '13, ACM, New York, NY, USA (2013)
52. Shi, J., He, J., Zhu, H., Fang, H., Huang, Y., Zhang, X.: ORIENTAIS: Formal verified OSEK/VDX real-time operating system. In: 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems. pp. 293–301 (Jul 2012)
53. Shi, J., Zhu, L., Fang, H., Guo, J., Zhu, H., Ye, X.: xBIL – a hardware resource oriented binary intermediate language. In: 2012 IEEE 17th International Conference on Engineering of Complex Computer Systems. pp. 211–219 (Jul 2012)
54. Slind, K., Norrish, M.: A brief overview of HOL4. In: International Conference on Theorem Proving in Higher Order Logics. pp. 28–32. Springer (2008)
55. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper. Hyderabad, India (Dec 2008)
56. Tan, J., Tay, H.J., Gandhi, R., Narasimhan, P.: Auspice: Automatic safety property verification for unmodified executables. In: VSSTE. pp. 202–222. Springer (2015)
57. Verbeek, F., Bockenek, J.A., Ravindran, B.: Artifact – Highly automated formal proofs over memory usage of assembly code (2020). [10.5281/zenodo.3676687](https://doi.org/10.5281/zenodo.3676687)
58. Wang, F., Shoshitaishvili, Y.: Angr – the next generation of binary analysis. In: 2017 IEEE Cybersecurity Development (SecDev). pp. 8–9. IEEE (2017)
59. Wenzel, M.: Isabelle/Isar—a generic framework for human-readable proof documents. From Insight to Proof—Festschrift in Honour of Andrzej Trybulec **10**(23), 277–298 (2007)
60. Yu, Y.: Automated Proofs of Object Code for a Widely Used Microprocessor. Ph.D. thesis, University of Texas at Austin (1992)
61. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). pp. 337–352 (2013)

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

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





GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts * †

Elvira Albert^{1,2}, Jesús Correas², Pablo Gordillo²,
Guillermo Román-Díez³, and Albert Rubio^{1,2}



¹ Instituto de Tecnología del Conocimiento, Spain
² Complutense University of Madrid, Spain
³ Universidad Politécnica de Madrid, Spain

Abstract. We present the main concepts, components, and usage of GASOL, a Gas Analysis and Optimization tool for Ethereum smart contracts. GASOL offers a wide variety of *cost models* that allow inferring the gas consumption associated to selected types of EVM instructions and/or inferring the number of times that such types of bytecode instructions are executed. Among others, we have cost models to measure only storage opcodes, to measure a selected family of gas-consumption opcodes following the Ethereum’s classification, to estimate the cost of a selected program line, etc. After choosing the desired cost model and the function of interest, GASOL returns to the user an upper bound of the cost for this function. As the gas consumption is often dominated by the instructions that access the storage, GASOL uses the gas analysis to detect under-optimized storage patterns, and includes an (optional) automatic optimization of the selected function. Our tool can be used within an Eclipse plugin for Solidity which displays the gas and instructions bounds and, when applicable, the gas-optimized Solidity function.

1 Introduction and Main Applications

Ethereum [27] is a global, open-source platform for decentralized applications that has become the world’s leading programmable blockchain. As other blockchains, Ethereum has a native cryptocurrency named *Ether*. Unlike other blockchains, Ethereum is programmable using a Turing complete language, i.e., developers can code smart contracts that control digital value, run exactly as programmed, and are immutable. A smart contract is basically a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. Smart contracts on the Ethereum blockchain are metered using *gas*. Gas is a unit that measures the amount of computational effort that it will take to execute each operation. Every single operation in Ethereum, be it

*This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, the MINECO and FEDER (EU) projects TIN2015-69175-C4-2-R and TIN2015-69175-C4-3-R, by the CM projects P2018/TCS-4314 and S2018/TCS-4339 co-funded by EIE Funds of the EU and by the UCM CT27/16-CT28/16 grant.

†The software and dataset used during the current study are available at [10.6084/m9.figshare.11876697](https://doi.org/10.6084/m9.figshare.11876697)

a transaction or a smart contract instruction execution, requires some amount of gas. The gas consumption of the Ethereum Virtual Machine (EVM) instructions is spelled out in [27]; importantly, instructions that use replicated storage are gas-expensive. Miners get paid an amount in *Ether* which is equivalent to the total amount of gas it took them to execute a complete operation. The rationale for gas metering is threefold: (i) Paying for gas at the moment of proposing the transaction prevents the emitter from wasting miners computational power by requiring them to perform worthless intensive work. (ii) Gas fees disincentive users to consume too much of replicated *storage*, which is a valuable resource in a blockchain-based consensus system (this is why storage bytecodes are gas-expensive). (iii) It puts a cap on the number of computations that a transaction can execute, hence prevents DoS attacks based on non-terminating executions.

Solidity [13] is the most popular language to write Ethereum smart contracts that are then compiled into EVM bytecode. The Solidity compiler, `solc`, is able to generate only *constant* gas bounds. However, when the bounds are *parametric* expressions that depend on the function parameters, on the contract state, or on the blockchain state (according to the experiments in [8] this happens in almost 10% of the functions), named `solc`, returns ∞ as gas bound. This paper presents GASOL [6], a resource analysis and optimization tool that is able to infer parametric bounds and optimize the gas consumption of Ethereum smart contracts. GASOL takes as input a smart contract (either in EVM, disassembled EVM, or in Solidity source code), a selection of a cost model among those available in the system (c.f. Section 2), and a selected public function, and it automatically infers *cost upper bounds* for this function. Optionally, the user can enable the gas optimization option (c.f. Section 3) to optimize the function w.r.t. storage usage, a highly valuable resource. GASOL has a wide range of applications: (1) It can be used to estimate the gas fee for running transactions, as it soundly over-approximates the gas consumption of functions. (2) It can be used to certify that the contract is free of out-of-gas vulnerabilities, as our bounds ensure that if the gas limit paid by the user is higher than our inferred gas bounds, the contract will not run out-of-gas. (3) As an attacker, one might estimate, how much *Ether* (in gas), an adversary has to pour into a contract in order to execute an out-of-gas attack. Also, attacks were produced by introducing a very large number of underpriced bytecode instructions [23]. Our cost models could allow detecting these second type of attacks by measuring how many instructions will be executed (that should be very large) while its associated gas consumption remains very low. (4) As we will show in the paper, the gas analysis can be used to detect gas-expensive fragments of code and automatically optimize them.

2 Gas Analysis using Gasol

Figure 1 overviews the components of the GASOL tool [6]. The programmer can use GASOL during the software development process from its Eclipse plugin that allows selecting the cost model of interest and the function to be analyzed and/or optimized from the Outline. This selection together with the compiled EVM code is sent to the gas analyzer. A technical description of all phases

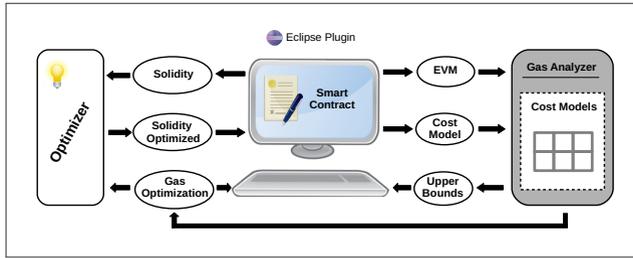


Fig. 1. Overview of GASOL's components

that comprise a gas analysis for EVM smart contracts is given in [8]. Basically, the analyzer uses various tools [3,7] to extract the CFGs and decompile them into a high-level representation from which upper bounds (UB) are produced by using extensions of resource analyzers and solvers [4,5]. However, in our basic gas analyzer named GASTAP [8], there was only one cost model to compute the overall gas consumption of the function (including the opcode and memory gas costs [27]), while GASOL is an extension of GASTAP that introduces optimization, a wide variety of analysis options to define novel cost models, and an Eclipse plugin. The UBs are provided to the user in the console as well as in markers for functions within the Eclipse editor. If the user had selected the optimization option, the analyzer detects potential sources of optimization and feeds them to the optimizer to generate an optimized Solidity function within a new file.

Fig. 2 displays our Eclipse plugin that contains a fragment of the public smart contract `ExtraBalToken` [1] used as running example. We can see its six state variables and its function fill that we will analyze and optimize. The right side window shows GASOL's configuration options to set up the *cost model*:

(i) *Type of resource (gas/instructions)*: by selecting *gas*, we estimate the gas consumption according to the gas model in [27] (hence, use GASOL as a gas analyzer); by selecting *instructions*, we estimate the number of bytecode instructions executed (using GASOL as a standard complexity analyzer).

(ii) *Type of instructions*: allows selecting which instructions (or group of instructions) will be measured as follows.

- *All*: every bytecode instruction will be measured. For instance, by selecting *gas* in (i), the function fill, and this option, we obtain as gas bound: $1077 + 40896 \cdot data$. Besides, by using this option, GASOL also yields the so-called memory gas (see[27]): $3 \cdot (data+5) + \left\lfloor \frac{(data+5)^2}{512} \right\rfloor$. The analyzer abstracts arrays by their length, hence, these bounds are functions of the *length of the input array* (denoted as *data*) and can be used, e.g., to determine precisely how much gas is necessary to run a transaction that executes this function.

- *Gas-family*: [27] classifies bytecode instructions according to their gas consumed in six groups: zero, base, verylow, low, mid and high. Instructions that do not belong to any of the previous groups are considered as single families. This option provides the cost due to each gas-family separately and, by using the filter in (iii), we can type the name of the desired group(s). As an example, for the function fill using *gas* in (i), we obtain gas bounds $297 + 315 \cdot data$ and $16 + 8 \cdot data$ for the gas-families *verylow* and *mid*, resp.

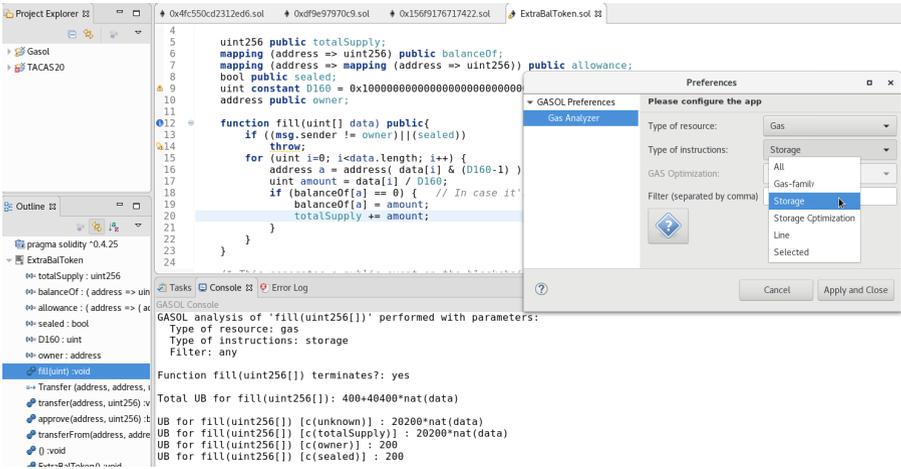


Fig. 2. Excerpt of smart contract `ExtraBalToken` in Solidity within Eclipse plugin.

- *Storage*: only the instructions that access the storage (namely bytecodes `SLOAD` and `SSTORE`) are accounted. The gas bounds displayed within the Eclipse console in Fig. 2 correspond to this setting, where we can see that the gas due to the access of each basic storage variable is shown separately. The first row `unknown` accumulates the gas of all accesses to non-basic types (data structures) as we still cannot identify them. By comparing this storage gas with the overall gas bound shown above for *All*, we can observe that most of the gas consumed by the function is indeed dominated by the storage (namely 40.000 out of 40.896 at each loop iteration) and it is thus a target for optimization (see Sec. 3).
- *Storage-optimization*: it bounds the number of `SLOAD` and `SSTORE` instructions executed by the current function (excluding those in transitive calls). It is the cost model that is used to detect and carry out the optimization described in Sec. 3. Thus, it is the only selection that enables the *Gas optimization* that appears as third option, and forces the selection of “instructions” as type of resource in (i). We obtain for the state variable `totalSupply` the bound: $2 \cdot data$, which captures that we execute two accesses (one read, one write) to field `totalSupply` at each loop iteration.
- *Line*: this option allows specifying the line number (of the Solidity program) whose cost will be measured, and the remaining lines will be filtered out. For instance, if the line number specified in the filter (iii) is 17, i.e., the Solidity instruction: `uint amount = data[i]/D160`, the obtained gas bound is $3 + 97 \cdot data$. In the absence of number in the filter, the bounds are given separately for all program lines. This option is intended to help the programmer in improving the gas consumption of her code by trying out different implementation options and comparing the results.
- *Selected*: allows computing the consumption associated to each different EVM instruction separately. For instance, if we select the bytecode instructions `MLOAD` and `SHA3`, we obtain the gas bounds $6 + 15 \cdot data$ and $84 \cdot data$ resp. As in the previous option, the filter allows the user to select the instructions of interest and filter out the remaining.

(iii) *Filter*: this is a text field used to filter out information from the UBs. For *gas-family*, the user can specify low, mid, etc. For *storage*, it allows specifying the name of the basic field(s) whose storage will be measured. For *line* and *selected*, we can type the line numbers and names of bytecode instructions of interest. Once all options have been selected, we have set up a cost model that is sent together with the EVM code to the gas analyzer and, after analysis, it outputs an UB for the selected function w.r.t. the cost model activated by the options. This UB is displayed, as shown in Fig. 2 in the console of the Eclipse plugin, and also within markers next to the function definition.

3 Gas Optimization using Gasol

The information yield by the gas analysis is used in GASOL to detect potential optimizations. Currently, the optimization target is the reduction of the gas consumption associated to the usage of storage. In particular, we aim at replacing multiple accesses to the same (global) storage data within a fragment of code (each write access costs 20.000 in the worst case and 5.000 in the best case) by one access that copies the data in storage to a (local) memory position followed by accesses to such memory position (an access to the local memory costs only 3) and a final update to the storage if needed. The cost model number of instructions for storage-optimization described in Sec. 2 allows us to detect such storage optimizations, namely for each different field, if we get a bound that is different from one, we know that there may be multiple accesses to the same position in the storage and we try to replace them by gas-efficient memory accesses. Our transformation is done at the level of the Solidity code, by defining a local variable with the same name as the state variable to transform, and introducing setter and getter functions to access the storage variable. Currently, we can transform accesses to variables of basic types, in the future, we plan to extend it to data structures (maps and arrays). The number of instructions bound for field `totalSupply` is $2 \cdot data$ (hence $\neq 1$), and our optimization of `fill` is:

```

1  function fill (uint [] data) {
2      uint256 totalSupply = get_field_totalSupply ();
3
4      if ((msg.sender != owner)||!(sealed))
5          throw;
6      for (uint i=0; i<data.length; i++) {
7          address a = address( data[i] & (D160-1) );
8
9          uint amount = data[i] / D160;
10         if (balanceOf[a] == 0) {
11             balanceOf[a] = amount;
12             totalSupply += amount;
13         }
14         set_field_totalSupply (totalSupply);
15     }

```

The gas bound (using the option *All*) for the optimized `fill` yield by GASOL is $21368 + 20674 \cdot data$, which means that, assuming the worst case for write access to storage, the gas consumed inside the loop is 49.45% smaller than the one for the original `fill` function (the memory gas does not change). Note that, even if we consider the best case of 5.000 for write access to storage for the accesses we have optimized, the gas reduction is still around 20%. This is, in fact, what we have manually estimated using the actual data of the 82 times this function has been executed in the Ethereum blockchain, achieving with GASOL a total saving of almost 60M gas. As our transformation is local to the function, in order to be sound, we check that the transformed global data is not being accessed by

transitive calls. For instance, if there was a call to another function from function `fill` that accesses `totalSupply`, we would not transform it. Besides, for efficiency, we check if all accesses are read (bytecode `SLOAD`) and, in such case, we do not need to invoke the setter at the end (and avoid an unnecessary write access).

4 Related Tools and Conclusions

Numerous tools are being developed to catch different types of vulnerabilities of smart contracts [20,16,22,19,17,26,18,10,15,9]. As mentioned in Sec. 1, the Solidity compiler `solc` is not able to give any gas estimation for the running example, as its gas consumption is not constant. Therefore, new gas analysis tools are being developed to detect potential gas related vulnerabilities and to infer bounds in these complex situations. The purpose of the GASPER and MADMAX tools is precisely the detection of gas related vulnerabilities. MADMAX [14] focuses on identifying control- and data-flow patterns inherent for the gas-related vulnerabilities, thus, it works as a bug-finder, rather than as a gas analyzer like GASOL. Similarly, GASPER identifies gas-costly programming patterns [12] by matching specific control-flow patterns and using SMT solvers and symbolic computation. Thus, it is an optimization detector, not an automatic optimizer as GASOL. The recently developed `ebso` tool [24] also aims at optimizing the gas consumption of EVM code. In contrast to GASOL, `ebso`'s optimizations are limited to a basic block level, while our transformation might involve several blocks of the CFG and would not be achievable by `ebso`'s approach. Also, `ebso` is not guided by the results of an automatic resource analysis which can capture the expensive storage patterns as in our case. Instead it is based on a full exploration of all possible alternative instructions (within the considered block) that would lead to the same result and consume less gas. They have obtained a number of rewrite rules that define sequences of bytecode instructions that can be replaced by equivalent ones that consume less. We could easily incorporate such basic block replacement optimizations within our tool, and it is part of our agenda.

The approach of [21], like ours, aims at inferring precise gas bounds. Their approach is based on symbolically enumerating all execution paths [11] and unwinding loops to a limit. Instead, using resource analysis, GASOL infers the maximal number of iterations for loops and generates accurate gas bounds which are valid for any possible execution of the function and not only for the unwound paths. The approach by Marescotti *et al.* has not been implemented in the context of EVM and a tool like GASOL has not been delivered. An orthogonal line of work with ours is the construction of resource-oriented attacks [23] that exploit the weaknesses of the EVM gas model. GASOL's cost models could help detect this resource-oriented attacks by estimating the number of executed bytecode instructions (very high) and their associated gas consumption (very low).

Finally, there is a tendency to define new languages (see Scilla [25], Michelson [2]) for programming smart contracts that provide certain safety guarantees, e.g., Scilla [25] provides predictable gas consumption by disallowing general recursion and while-loops. However, Ethereum is today the most widely used blockchain, and Solidity the most popular programming language to write Ethereum smart contracts, for which a gas analyzer+optimizer is of clear relevance.

References

1. ExtraBalToken contract. <https://etherscan.io/address/0x5c40ef6f527f4fba68368774e6130ce6515123f2>
2. The Michelson Language. <https://www.michelson-lang.com>
3. Oyente: An Analysis Tool for Smart Contracts (2018), <https://github.com/melonproject/oyente>
4. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martín-Martín, E., Puebla, G., Román-Díez, G.: SACO: Static Analyzer for Concurrent Objects. In: TACAS. LNCS, vol. 8413, pp. 562–567. Springer (2014)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: SAS. LNCS, vol. 5079, pp. 221–237. Springer (2008)
6. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts (Artifact) (2020), Figshare 2020, [10.6084/m9.figshare.11876697](https://figshare.com/articles/GASOL_Gas_Analysis_and_Optimization_for_Ethereum_Smart_Contracts/11876697)
7. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In: ATVA. LNCS, vol. 11138, pp. 513–520. Springer (2018)
8. Albert, E., Gordillo, P., Rubio, A., Sergey, I.: Running on Fumes: Preventing Out-Of-Gas vulnerabilities in Ethereum Smart Contracts using Static Resource Analysis. In: VECoS. LNCS, vol. 11847, pp. 63–78. Springer (2019)
9. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In: CPP. pp. 66–77. ACM (2018)
10. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguélin, S.: Formal verification of smart contracts: Short paper. In: PLAS. pp. 91–96. ACM (2016)
11. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
12. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: SANER. pp. 442–446. IEEE Computer Society (2017)
13. Ethereum: Solidity (2018), <https://solidity.readthedocs.io>
14. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. PACMPL **2**(OOPSLA), 116:1–116:27 (2018)
15. Grishchenko, I., Maffei, M., Schneidewind, C.: A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In: POST. LNCS, vol. 10804, pp. 243–269. Springer (2018)
16. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzy, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. PACMPL **2**(POPL), 48:1–48:28 (2018)
17. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: NDSS. The Internet Society (2018)
18. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting The Laws of Order in Smart Contracts. CoRR **abs/1810.11605** (2018)
19. Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: USENIX Security Symposium. pp. 1317–1333. USENIX Association (2018)

20. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS. pp. 254–269. ACM (2016)
21. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing Exact Worst-Case Gas Consumption for Smart Contracts. In: ISoLA. LNCS, vol. 11247, pp. 450–465. Springer (2018)
22. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: ACSAC. pp. 653–663. ACM (2018)
23. Pérez, D., Livshits, B.: Broken metre: Attacking resource metering in EVM. CoRR [abs/1909.07220](https://arxiv.org/abs/1909.07220) (2019), <http://arxiv.org/abs/1909.07220>
24. Schett, M., Nagele, J.: Blockchain superoptimizer. In: 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019) (2019)
25. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with Scilla. In: 34th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2019) (2019)
26. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: CCS. pp. 67–82. ACM (2018)
27. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2014)

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

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





CPU Energy Meter: A Tool for Energy-Aware Algorithms Engineering

Dirk Beyer  and Philipp Wendler 

LMU Munich, Germany



Abstract. Verification algorithms are among the most resource-intensive computation tasks. Saving energy is important for our living environment and to save cost in data centers. Yet, researchers compare the efficiency of algorithms still in terms of consumption of CPU time (or even wall time). Perhaps one reason for this is that measuring energy consumption of computational processes is not as convenient as measuring the consumed time and there is no sufficient tool support. To close this gap, we contribute CPU ENERGY METER, a small tool that takes care of reading the energy values that Intel CPUs track inside the chip. In order to make energy measurements as easy as possible, we integrated CPU ENERGY METER into BENCHEXEC, a benchmarking tool that is already used by many researchers and competitions in the domain of formal methods. As evidence for usefulness, we explored the energy consumption of some state-of-the-art verifiers and report some interesting insights, for example, that energy consumption is not necessarily correlated with CPU time.

Keywords: Energy Measurement · RAPL · Benchmarking · BenchExec

1 Introduction

There is a strong demand to save electrical energy, of which nowadays a large portion is used by computational processes. Most importantly, we need to protect the environment that we live in, but we also need to consider that energy usage is one of the most important cost factors in data centers: after computing devices are purchased and installed, the operational cost is dominated by the cost of consumed electrical energy. And since most of the used electrical energy is turned into heat energy, there is follow-up cost for the cooling system, which sets the limits of used energy for each rack in a data center [16].

In order to control energy consumption, we first need to measure it. Work in the area of green software engineering identified a lack of data and insufficient tool support [12]. Energy consumption of an algorithm is often reduced to CPU time, which seems to be a natural choice at a first look, but after more accurate measurement we know that this reduction leads to wrong conclusions.

Why is energy usage of verification algorithms not measured but only CPU time? Most likely it is technically too difficult for researchers to measure energy consumption, because it would require external hardware that is not common or because internal energy measurements are not well-known and complex to use.

© The Author(s) 2020

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12079, pp. 126–133, 2020.

https://doi.org/10.1007/978-3-030-45237-7_8

In order to provide a solution to this problem, we contribute an open-source lightweight tool that enables convenient energy measurement for a large range of modern CPUs. The tool CPU ENERGY METER makes it easy and convenient to access energy measurements done by the CPU for various of its parts. Furthermore, we integrate energy measurement in the benchmarking framework BENCHEXEC, which is widely used by researchers and competitions (e.g., [2]).

Using CPU ENERGY METER does not require any extra hardware, but accesses the existing feature for energy measurement called RAPL that Intel CPUs provide. This convenience comes with a limitation: We can only access measurement values for those parts of the computing board that the CPU measures, but no external equipment, such as hard drives and the power supply itself.

Related Work. Energy measurements should be used for algorithm engineering [1], and there is a strong need for tool support, such as POWERPACK [8]. RAPL is being studied as a measurement method for energy consumption [6, 9, 10, 13, 17], and energy measurements that are based on RAPL are being developed for specific scenarios [11, 15, 18, 19] and used to evaluate algorithms [7]. CPU ENERGY METER makes energy measurement conveniently accessible to verification researchers. The most closely related project is the Performance API (PAPI) analysis library, which also supports RAPL [19], but this is a large library with a much larger scope than just energy measurements. In contrast, our tool is a ready-to-use solution for energy measurements that is easy to install and use.

2 Intel Running Average Power Limit (RAPL)

The Intel Running Average Power Limit (RAPL) [14] is a feature of Intel CPUs that allows to measure and limit the energy consumption of CPUs. It is available since the 2nd generation of the Intel Core architecture (code name “Sandy Bridge”), i.e., on Intel Core i3/i5/i7 2000 and newer, as well as Intel Xeon E3/E5/E7 CPUs. This covers a wide range of common CPUs for notebooks, desktops, and servers.

One part of RAPL consists of access to a series of hardware counters in which the CPU accumulates the energy it has consumed. RAPL supports measuring the energy consumption of so-called “domains”, and up to five domains are supported by current CPUs: package, PP0, PP1, DRAM, and PSYS. Which hardware units are included in which domain is not clearly specified by Intel, but in general we can use the following assumption: The package domain refers to the whole CPU, the PP0 domain refers to the processor cores, and the PP1 domain refers to other units such as an integrated graphics unit. The domains DRAM and PSYS may provide information on the energy consumption of the RAM and other hardware on the mainboard, but both need special support from the hardware platform and its values may not be comparable between different systems.

There is no official information by Intel on the precision of the measurements except that the counters are updated approximately every 1 ms. The resolution of the values varies between the CPUs, but is typically $\frac{1}{2^{16}}$ J or $\frac{1}{2^{14}}$ J, i.e., in the order of 10^{-5} J. For the first generation of CPUs with RAPL, the energy consumption was approximated by the CPU and imprecise, but for subsequent generations the precision had been improved [6, 7, 10].

3 CPU ENERGY METER

Our tool CPU ENERGY METER provides access to the energy-measurement features of Intel CPUs to users. It was developed based on the tool Intel Power Gadget for Linux¹. Our tool is available as open source under the permissive 2-clause BSD license and hosted on GitHub². Installation packages of CPU ENERGY METER are available for Debian-based distributions (e.g., Ubuntu).

CPU ENERGY METER measures the energy consumption of the CPU(s) of a system for a specific time interval as reported by the RAPL interface (cf. Sect. 2). In order to ensure the highest possible measurement precision with the lowest possible overhead, it reads the RAPL energy counters as rarely as possible instead of using continuous sampling, while at the same time reading the counters often enough to safely detect and account for counter overflows. Furthermore, our tool was developed to use a minimal amount of necessary dependencies and permissions in order to make its installation as easy as possible.

Requirements. CPU ENERGY METER requires a system with one or more Intel CPUs that support the RAPL feature. It needs direct access to the CPUs, thus running in a virtual machine is not supported. Accessing the model-specific registers of CPUs with the energy measurements is done via the Linux kernel module `msr`³, which needs to be loaded and provides device files named `/dev/cpu/*/msr`.

Typically, access to these device files is granted only to the user `root`. In order to not need to execute CPU ENERGY METER as `root`, one can change the file permissions of the device files appropriately (e.g., by granting read permissions to a group `msr` and making CPU ENERGY METER always execute as this group using the “`setgid`” permission). Furthermore, CPU ENERGY METER needs the capability `CAP_SYS_RAWIO`⁴, which can be granted using `setcap`⁵. The installation packages of CPU ENERGY METER attempt to automatically configure the system such that every user can execute the tool without granting any other non-standard permissions to users. In any case (whether executed as `root` or not), CPU ENERGY METER drops all unnecessary permissions as soon as possible using the library “`libcap`”⁶ in order to reduce any risk related to the non-standard permissions.

Usage. CPU ENERGY METER is intended primarily to be used by benchmarking frameworks, however, manual execution is also possible. When the tool is executed, it starts the measurements and prints the consumed energy for all supported domains and CPUs of the system as soon as it is killed via the interrupt signal or `Ctrl+C`. Intermediate measurements are printed when the signal `USR1` is received. To manually measure the energy consumption of the duration of a specific command, one can execute the following command line, for example:

```
cpu-energy-meter & some_command ; kill -INT %1
```

¹ <https://software.intel.com/en-us/articles/intel-power-gadget>

² <https://github.com/sosy-lab/cpu-energy-meter>

³ <http://man7.org/linux/man-pages/man4/msr.4.html>

⁴ <http://man7.org/linux/man-pages/man7/capabilities.7.html>

⁵ <http://man7.org/linux/man-pages/man8/setcap.8.html>

⁶ <https://sites.google.com/site/fullycapable/>

This will measure the energy consumption of all CPUs during the whole time that the specified command is running, regardless of whether this energy consumption is caused by the specified command or by other processes running in parallel (this is a limitation of the RAPL feature). Thus, measuring the energy consumption during a specific time period (e.g., 10s) can be done by replacing `some_command` with `sleep 10`.

The output values are given with the unit Joule, and can be formatted either in a way that is optimized for being read by humans (cf. Fig. 1) or parsed by programs.

```
+-----+
| CPU Energy Meter  Socket 0 |
+-----+
Duration           9.990624 sec
Package            15.898926 Joule
Core               1.695740 Joule
Uncore            0.352661 Joule
DRAM              11.024841 Joule
PSYS              104.778931 Joule
```

Fig. 1: Example output of CPU ENERGY METER on a single-CPU system of the SkyLake generation (with all five domains supported)

Integration into BENCHEXEC. We have contributed an integration of CPU ENERGY METER into the benchmarking framework BENCHEXEC [4], because BENCHEXEC is widely used in the formal-methods community (e.g., SV-COMP [2]). Starting with version 1.16, BENCHEXEC automatically executes CPU ENERGY METER if the latter is installed, and it reports the energy results in the same manner as the results of its internal time and memory measurements (BENCHEXEC supports the creation of CSV tables and interactive HTML tables with plots for its benchmarking results). BENCHEXEC will report the energy consumption only if all cores of one or more CPUs are used for each tool execution, because we cannot distinguish between the energy consumption of individual processes.

4 Applications

The 8th International Competition on Software Verification (SV-COMP’19) [3] measured energy consumption of verification tools using BENCHEXEC and CPU ENERGY METER and for the first time provided an alternative “green” ranking based on energy efficiency (CPU-energy usage divided by achieved score). This ranking was indeed considerably different from the main score-based ranking, with no overlap between the top three green verifiers and the top three verifiers in the category “C-Overall”. Furthermore, the winner in the green ranking is two orders of magnitude more efficient than the last tool in the ranking (64 J per score point vs. 4200 J per score point). This shows an enormous potential of efficiency improvements and energy savings if verification researchers get access to easy measurements of energy usage.

In the following, we analyze in more detail some energy measurements of SV-COMP’19, which provides all raw results online⁷. We pick the results for the submissions CBMC⁸ and CPA-SEQ⁹ across all categories. CPA-SEQ is the winner of the category “C-Overall”, written in Java, and employs several different algorithms, some of which are partially parallelized. The garbage collector that

⁷ <https://sv-comp.sosy-lab.org/2019/results/results-verified/All-Raw.zip>

⁸ <http://www.cprover.org/cbmc/> ⁹ <https://cpatchecker.sosy-lab.org/>

Table 1: Selection of Energy Measurements from SV-COMP’19

RAPL domain	CBMC			CPA-SEQ		
	Package	PP0 (Core)	DRAM	Package	PP0 (Core)	DRAM
Average power used per task with regard to wall time (energy divided by wall time):						
Min (W)	1.9	1.2	0.63	4.4	3.4	1.6
Max (W)	25	24	5.5	36	35	7.2
Avg (W)	9.7	8.8	2.4	20	19	2.8
Std. Dev. (W)	3.2	3.2	0.71	6.2	6.2	0.48
Average power used per task with regard to CPU time (energy divided by CPU time):						
Min (W)	1.8	1.1	0.58	4.2	3.2	0.70
Max (W)	23	22	5.5	17	16	6.8
Avg (W)	9.6	8.7	2.4	9.6	9.0	1.5
Std. Dev. (W)	3.1	3.1	0.74	1.8	1.7	0.60

is used by the JVM adds some more parallelism. CBMC is written in C++ and uses bounded model checking in a strictly sequential implementation. Thus, we expect that the energy consumption of these tools has different characteristics. SV-COMP’19 executed both tools for 10 522 tasks (CPU-time limit 900s per task, Intel Xeon E3-1230 v5 CPU, quad-core with hyper-threading, 3.4 GHz, all 8 processing units of the CPU and 15 GB of memory were available to each tool execution, Ubuntu 18.04 64-bit with Linux kernel 4.15 was the operating system).

We now compare the energy consumption of the RAPL domain “Package” with the CPU time for CBMC in Fig. 2 and for CPA-SEQ in Fig. 3.¹⁰ In the plot, all results that lie on the same line through the origin belong to tool executions for which the energy consumption per second of CPU time (in $\frac{J}{s} = W$) was the same (this would be the average power of the CPU if measuring wall time instead of CPU time). We provide additional statistics in Table 1 and two graphs that compare the CPU time and the energy consumption of the two tools in Fig. 4.

Insight: *Also for verification tools, high values for CPU time do not imply high values for energy.* Figure 2 has a large vertical area of data points where the CPU time is close to the time limit. For those verification runs, the energy is in the range of 2.0 kJ to 15 kJ. This shows that for a specific CPU time, the energy consumption (and average power, cf. Table 1) for different verification tasks can vary by a factor of 7.

Insight: *Comparing different verification tools regarding CPU time can lead to different conclusions than energy-based comparisons.* The graph on the left of Fig. 4 compares CBMC and CPA-SEQ regarding CPU time, the graph on the right compares them regarding energy consumption. The difference between the shapes of these two graphs shows that looking at the energy consumption when comparing tools is an interesting addition to comparing only CPU time, and that

¹⁰ For CPA-SEQ, the CPU time is sometimes higher than 900s because SV-COMP lets tools optionally run for more than the time limit in order to print additional statistics (but any result after the time limit is of course discarded).

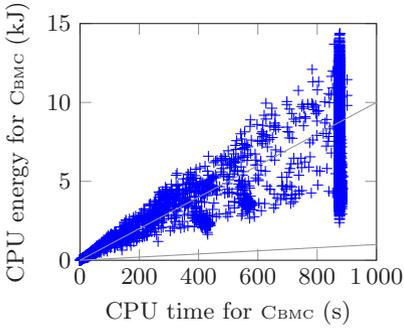


Fig. 2: Comparison of CPU time vs. energy consumption for CBMC

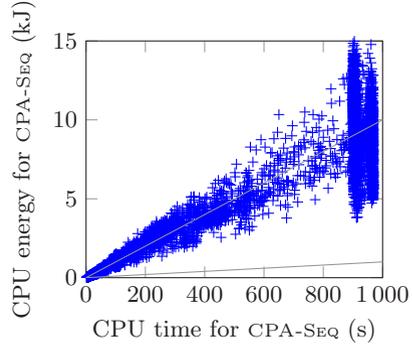


Fig. 3: Comparison of CPU time vs. energy consumption for CPA-SEQ

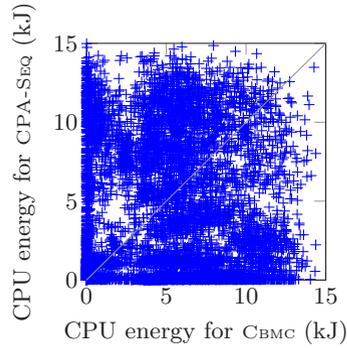
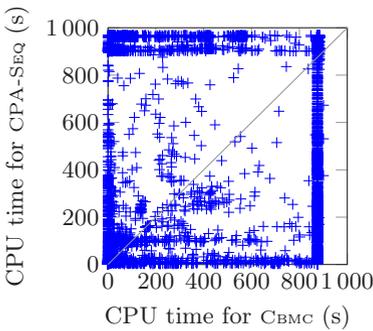


Fig. 4: Comparison of CBMC and CPA-SEQ with regard to CPU time and energy

the similar statistics on power usage with regard to CPU time (cf. lower part of Table 1 and Figs. 2 and 3) can be misleading; if the power-usage characteristics of both tools were the same, the two graphs in Fig. 4 would look similar.

5 Conclusion

Verification algorithms consume large amounts of energy and thus, it is prohibitive to ignore the energy characteristics of algorithms when comparing their quality. Although this matter is understood, the verification community does not measure energy. We believe that this is because measurement of energy is complex and requires a lot of additional effort. The lightweight tool CPU ENERGY METER fills this gap: It supports reading Intel-RAPL-based energy measurements in a convenient way and—via integration into BENCHEXEC—using a tool environment that many verification researchers use anyway already.

An analysis of a large data set from a verification competition invalidates a wide-spread assumption: the data quickly reveal that energy consumption can deviate significantly from the consumed CPU time. Thus, it is not sufficient to measure CPU time.

Data Availability Statement. A replication package for this article including CPU ENERGY METER and BENCHEXEC is available at Zenodo [5]. Current versions of CPU ENERGY METER are available at <https://github.com/sosy-lab/cpu-energy-meter> and <https://doi.org/10.5281/zenodo.1300309>. The dataset from SV-COMP'19 [3] that was analyzed in Sect. 4 is available online at <https://sv-comp.sosy-lab.org/2019/results/results-verified/All-Raw.zip>.

References

1. Bekas, C., Curioni, A.: A new energy aware performance metric. *Computer Science - R&D* **25**(3-4), 187–195 (2010). <https://doi.org/10.1007/s00450-010-0119-z>
2. Beyer, D.: Reliable and reproducible competition results with BENCHEXEC and witnesses (Report on SV-COMP 2016). In: *Proc. TACAS*. pp. 887–904. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
3. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: *Proc. TACAS* (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
4. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer* **21**(1), 1–29 (2019). <https://doi.org/10.1007/s10009-017-0469-y>
5. Beyer, D., Wendler, P.: Replication package for article ‘CPU Energy Meter: A tool for energy-aware algorithms engineering’ in *Proc. TACAS ’20*. Zenodo (2020). <https://doi.org/10.5281/zenodo.3679337>
6. Desrochers, S., Paradis, C., Weaver, V.M.: A validation of DRAM RAPL power measurements. In: *Proc. Int. Symposium on Memory Systems (MEMSYS)*. pp. 455–470. ACM (2016). <https://doi.org/10.1145/2989081.2989088>
7. Dongarra, J.J., Ltaief, H., Luszczek, P., Weaver, V.M.: Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architectures. In: *Proc. Int. Conference on Cloud and Green Computing (CGC)*. pp. 274–281. IEEE (2012). <https://doi.org/10.1109/CGC.2012.113>
8. Ge, R., Feng, X., Song, S., Chang, H., Li, D., Cameron, K.W.: POWERPACK: Energy profiling and analysis of high-performance systems and applications. *IEEE Trans. Parallel Distrib. Syst.* **21**(5), 658–671 (2010). <https://doi.org/10.1109/TPDS.2009.76>
9. Hackenberg, D., Ilsche, T., Schöne, R., Molka, D., Schmidt, M., Nagel, W.E.: Power measurement techniques on standard compute nodes: A quantitative comparison. In: *Proc. Int. Symposium on Performance Analysis of Systems & Software (ISPASS)*. pp. 194–204. IEEE (2013). <https://doi.org/10.1109/ISPASS.2013.6557170>
10. Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., Geyer, R.: An energy efficiency feature survey of the Intel Haswell processor. In: *Proc. Int. Parallel and Distributed Processing Symposium (IPDPS)*. pp. 896–904. IEEE (2015). <https://doi.org/10.1109/IPDPSW.2015.70>
11. Hähnel, M., Döbel, B., Völp, M., Härtig, H.: Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review* **40**(3), 13–17 (2012). <https://doi.org/10.1145/2425248.2425252>
12. Hindle, A.: Green software engineering: The curse of methodology. *Tech. Rep. 4:e1470v2*, PeerJ PrePrints (2016). <https://doi.org/10.7287/peerj.preprints.1470v2>
13. Ilsche, T., Hackenberg, D., Graul, S., Schöne, R., Schuchart, J.: Power measurements for compute nodes: Improving sampling rates, granularity and accuracy. In: *Proc.*

- Int. Green and Sustainable Computing Conference (IGSC). pp. 1–8. IEEE (2015). <https://doi.org/10.1109/IGCC.2015.7393710>
14. Intel: Intel 64 and IA-32 architectures software developer’s manual, vol. 3B, chap. 14.9 (December 2017), available at <https://software.intel.com/en-us/articles/intel-sdm>
 15. Khan, K.N., Ou, Z., Hirki, M., Nurminen, J.K., Niemi, T.: How much power does your server consume? Estimating wall socket power using RAPL measurements. *Computer Science - R&D* **31**(4), 207–214 (2016). <https://doi.org/10.1007/s00450-016-0325-4>
 16. Scaramella, J., Eastwood, M.: Solutions for the data center’s thermal challenges. Tech. rep., IDC (2007), available at https://www-935.ibm.com/services/fr/igs/pdf/idc_opinion_coolblue_wp.pdf
 17. Schuchart, J., Hackenberg, D., Schöne, R., Ilsche, T., Nagappan, R., Patterson, M.K.: The shift from processor power consumption to performance variations: Fundamental implications at scale. *Computer Science - R&D* **31**(4), 197–205 (2016). <https://doi.org/10.1007/s00450-016-0327-2>
 18. Venkatesh, A., Kandalla, K.C., Panda, D.K.: Evaluation of energy characteristics of MPI communication primitives with RAPL. In: Proc. Int. Symposium on Parallel & Distributed Processing (IPDPSW). pp. 938–945. IEEE (2013). <https://doi.org/10.1109/IPDPSW.2013.243>
 19. Weaver, V.M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., Moore, S.: Measuring energy and power with PAPI. In: Proc. Int. Conference on Parallel Processing Workshops (ICPPW). pp. 262–268. IEEE (2012). <https://doi.org/10.1109/ICPPW.2012.39>

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

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



Logic and Proof



Practical Machine-Checked Formalization of Change Impact Analysis

Karl Palmkog¹, Ahmet Celik², and Milos Gligoric³

¹ KTH Royal Institute of Technology, Stockholm, Sweden

² Facebook, Seattle, WA, USA

³ The University of Texas at Austin, Austin, TX, USA
palmkog@kth.se, celik@fb.com, gligoric@utexas.edu

Abstract. Change impact analysis techniques determine the components affected by a change to a software system, and are used as part of many program analysis techniques and tools, e.g., in regression test selection, build systems, and compilers. The correctness of such analyses usually depends both on domain-specific properties and change impact analysis, and is rarely established formally, which is detrimental to trustworthiness. We present a formalization of change impact analysis with machine-checked proofs of correctness in the Coq proof assistant. Our formal model factors out domain-specific concerns and captures system components and their interrelations in terms of dependency graphs. Using compositionality, we also capture hierarchical impact analysis formally for the first time, which, e.g., can capture when impacted files are used to locate impacted tests inside those files. We refined our verified impact analysis for performance, extracted it to efficient executable OCaml code, and integrated it with a regression test selection tool, one regression proof selection tool, and one build system, replacing their existing impact analyses. We then evaluated the resulting toolchains on several open source projects, and our results show that the toolchains run with only small differences compared to the original running time. We believe our formalization can provide a basis for formally proving domain-specific techniques using change impact analysis correct, and our verified code can be integrated with additional tools to increase their reliability.

Keywords: Change impact analysis · Regression test selection · Coq.

1 Introduction

Change impact analysis aims to determine the components affected by a change to a software system, e.g., the modules or files affected by a modified line of code [3,4]. Change impact analysis techniques are used in many program analyses and tools, such as regression test selection (RTS) tools [26, 52, 59, 61], build systems [15, 21, 43, 45], and incremental compilers [48].

Change impact analysis techniques typically mix domain- and language-specific concepts, such as method call graphs and class files, with more abstract notions, such as dependencies, transitive closures, and topological sorts. This can

complicate reasoning about the correctness (*safety*) of a technique. For example, to the best of our knowledge, RTS techniques for Java-like languages have never been argued to be safe (i.e., to never omit tests affected by a change) by *machine-checked reasoning*—only by high-level pen-and-paper proofs [51, 55, 60].

In this paper, we present a formalization of key concepts used in many change impact analysis techniques—concepts that are *independent of any language or application domain*. Our formalization represents system components and their interrelations as vertices and edges in explicit *dependency graphs*. We consider whether components are *impacted* by changes between two system revisions by computing transitive closures of modified graph vertices in the *inverse* of the dependency graph from the old revision. This has been described as “invalidating the upward transitive closure” [14]. Among impacted vertices, we identify those that are *checkable*, representing, e.g., a test method, that can be re-executed.

We encoded our formal model as a library in the Coq proof assistant, and proved two key correctness properties: *soundness* and *completeness*. Soundness, intuitively, states that the outcomes of executing checkable vertices that are unimpacted in the new revision are the same as they would be in the previous revision. Completeness roughly states that all checkable vertices in the new revision are members of the set of all added, impacted, and unimpacted vertices.

Based on our correctness approach, we also defined and proved correct two strategies for *hierarchical* change impact analysis that are roughly analogous to, on the one hand, file-based incremental builds [43, 54], and on the other hand, hybrid regression test selection [46, 60]. To the best of our knowledge, hierarchical change impact analysis is previously unexplored in formal settings like ours. Ultimately, by proving some basic properties about relations between vertices and results of executing checkable vertices, developers can use our model and library to obtain end-to-end guarantees for domain-specific impact analyses.

To capture our model of system components and their dependencies in Coq, we used the Mathematical Components (MC) library [42] and its representation of relations, finite graphs, and subtypes [25, 28, 29]. For the formal proofs, we used the SSReflect proof language and followed the idiom of the MC library of leveraging boolean decision procedures in proofs via *small-scale reflection* [9, 30, 31]. To obtain efficient executable code, we performed several verified refinements of our initial Coq encoding. From our refined functions and datatypes, we then derived a practical tool, dubbed CHIP, by carefully extracting Coq code to OCaml and linking it with an assortment of OCaml libraries. CHIP can be viewed as a *verified component* for change impact analysis that can either be integrated into verified systems or used in conventionally developed systems.

To ensure the adequacy of our formal model, we performed an empirical study using CHIP. Specifically, we integrated CHIP with EKSTAZI [26], a tool for class-based regression test selection in Java, with ICOQ [11], a tool for regression proof selection in Coq itself, and with TUP [54], a build system similar to `make`, replacing the existing components for change impact analysis in all these tools. We then compared the outcome and running time between the respective modified and original tool versions when applied to the revision histories of several

open-source projects. This approach is along the lines of previous evaluations of formal specifications [8, 20, 33] and RTS techniques [26, 37, 60]. During our evaluation of CHIP, we also located and addressed several performance bottlenecks.

We make the following contributions:

- **Basic formal model:** We present a formalization of change impact analysis in terms of finite graphs and sets, encoded in the Coq proof assistant via the MC library. We formulated and proved in Coq key correctness requirements for our analysis, namely, soundness and completeness.
- **Hierarchical formal model:** We extended our model to capture two strategies for hierarchical change impact analysis, where higher-level components are implicitly tied to lower-level components, and proved them both correct.
- **Library:** Our Coq development forms a library of definitions and lemmas that can assist in formally proving various techniques based on change impact analysis, such as regression test selection for Java, correct inside Coq.
- **Optimizations:** We refined our verified Coq functions and data structures to significantly improve performance in practice of code extracted to OCaml.
- **Tool:** From our refined Coq code, we derived a verified executable tool in OCaml for change impact analysis, CHIP, by carefully leveraging Coq’s code extraction mechanism. CHIP can be used as a *verified component* for both regular and hierarchical change impact analysis in other tools. The CHIP code, compatible with Coq 8.9, MC 1.7, and OCaml 4.07, is publicly available [47].
- **Evaluation:** We integrated CHIP with a tool for regression test selection in Java projects, EKSTAZI, one regression proof selection tool for Coq itself, iCOQ, as well as one build system, TUP, and evaluated the resulting toolchains on several medium to large-scale open source projects.

2 Background

In this section, we give some brief background on change impact analysis and its applications, and on the Coq proof assistant.

2.1 Change Impact Analysis

Broadly, we consider change impact analysis as the activity of identifying the potential consequences of a change to a software system. Formulated in this way, change impact analysis is an old concern in software engineering [4], and remains an active research topic as part of techniques and tools [1, 34, 53]. In early work, Arnold posited computing transitive closures of statically derived program call graphs as the fundamental technique for change impact analysis [3]. However, later research argues that dynamic analysis can be more precise [36] and lead to faster dependency collection for use in future analyses [26]. Our work aims to capture general concepts used in both static and dynamic approaches [10, 38].

2.2 Regression Test Selection and Regression Proof Selection

Regression test selection (RTS) techniques optimize regression testing – running tests at each project revision to check correctness of recent changes – by deselecting tests that are not affected by the recent changes [50, 59]. Traditionally,

RTS techniques maintain for each test a set of code elements (e.g., statements, methods, classes) on which the test depends. When code elements are modified, change impact analysis is used to detect those tests that are potentially affected by the changes. Prior work has studied RTS for various programming languages (e.g., C, C++, and Java), built dependency graphs statically or dynamically, and used various granularities of code elements (e.g., statements, methods, and classes). The meaning of the dependency graph is language-specific, but if the graph is properly constructed, the change impact analysis is independent of the language. For example, EKSTAZI [26], a recent RTS tool for Java projects, builds and maintains Java class file dependency graphs dynamically, and when a class file is modified, EKSTAZI uses change impact analysis to select all test classes that depend, directly or indirectly, on the modified class.

Regression proof selection (RPS) is the analogue of RTS for formal proofs, which, similarly to tests, can take a long time to check. The RPS technique implemented in the ICoQ tool for Coq [12] uses *hierarchical* selection [11], where impacted files are used to locate impacted proofs to be checked.

2.3 Build Systems

The classic build system `make` uses file timestamp comparisons to decide whether a task defined in a *build script* should be run. Dependency graphs are implicitly defined by tasks depending on the completion of other tasks, or on certain files, as expressed in the build script. In contrast to test execution, build script task execution typically produces side effects in the form of new files, e.g., files with object code in ELF format. Modern build systems such as Bazel [5] and CloudMake [21, 27] can use other ways than timestamps to find modified files, e.g., comparing cryptographic hashes of files across revisions. Recent alternative build systems that aim to replace `make` include TUP [54] and Shake [43]; the former uses an explicit persisted dependency graph.

2.4 The Coq Proof Assistant and Mathematical Components

Coq consists of, on the one hand, a small and powerful purely functional programming language, and on the other hand, a system for specifying properties about programs and proving them [6]. Coq is based on a constructive type theory [17, 18] which effectively reduces proof checking to type checking, and puts programming on the same foot as proving. Mathematical Components (MC) [42] is an extensive Coq library that provides many structures from mathematics, including finite sets, relations, and subtypes; we use the module `fingraph`, which was derived from Gonthier’s proof of the four-color theorem [28].

Datatypes and functions verified inside Coq to have some correctness property can be *extracted* to a practical programming language such as OCaml [40], and then integrated with libraries; extraction is used in several large-scale software verification projects [39, 57]. Obtaining efficient programs via extraction may require significant engineering because of discord between the requirements for formal correctness and agreeable program runtime behavior [19]. When target languages lack fully formal semantics, as is the case for OCaml, extraction cannot be fully trusted, but empirical evaluations are nevertheless encouraging [24, 58].

3 Formal Model

This section introduces our model, assumptions, and correctness approach.

3.1 Definitions

Components: Our model of change impact analysis uses two finite sets of vertices V and V' , where $V \subseteq V'$. Members of these sets represent the components of a system (e.g., files or classes) before and after a change, respectively.

Artifacts: We let A be a set of *artifacts*. An artifact is intended to be a concrete underlying representation of a component, e.g., an abstract syntax tree or the content of a file. We assume that the equality of two artifacts is decidable, i.e., that we can compute for all $a, a' \in A$ whether $a = a'$ or $a \neq a'$. To associate vertices with artifacts, we use two total functions $f: V \rightarrow A$ and $f': V' \rightarrow A$. In practice, we expect these functions to map vertices to compact *summaries* of component representations, such as checksums computed by cryptographic hash functions. Whenever $f(v) \neq f'(v)$ for some $v \in V$, we say that the artifact for v is *modified* after the revision; otherwise, it is *unmodified*.

Graphs: Let g be a binary relation on V . For $v, v' \in V$, we say that v *directly depends on* v' if $g(v, v')$ holds. For example, if v and v' represent classes in a Java-like language, v may be a subclass of v' . We will usually refer to relations like g as (dependency) *graphs*. We write g^{-1} for the inverse of g , i.e., we have $g^{-1}(v, v')$ iff $g(v', v)$. Moreover, we write $g^*(v, v')$ for when v and v' are transitively related in g , and say that v *transitively depends on* v' . We define the *reflexive-transitive closure* of a vertex $v \in V$ with respect to a graph g as the set $\{v' \mid g^*(v, v')\}$, i.e., as the set of all vertices reachable from v in g (which includes v itself).

Execution: We assume there is a subset $E \subseteq V'$ of *checkable* vertices, i.e., it is meaningful to apply some (side-effect free) function *check* on them and obtain some result. For example, a checkable vertex may represent a test method that either passes or fails when executed.

Impactedness: Let g be a dependency graph. We then say that a vertex $v \in V$ is *impacted* if it is reachable in g^{-1} from some modified vertex. Equivalently, v is impacted iff there is a $v' \in V$ such that $f(v') \neq f'(v')$ and $(g^{-1})^*(v', v)$. Additionally, a vertex $v'' \in V'$ is considered *fresh* whenever $v'' \notin V$.

We take the (disjoint) union of the set I of impacted vertices and the set F of fresh vertices, and consider the checkable vertices in this set, i.e., vertices in $(I \cup F) \cap E$. Intuitively, these are the only vertices that we need to consider in the new revision, since all other vertices in V' are *unimpacted*—and using *check* on unimpacted vertices will have the same outcome as in the old revision.

3.2 Example

Figure 1 illustrates the core idea of the graph-based change impact analysis approach we model. Figure 1(a) shows the original dependency graph, where, e.g., component 3 depends directly on components 1 and 2, and 5 depends directly on 3 and transitively on 1 and 2; dotted components are checkable. Figure 1(b) shows the inverse graph, with the modified component 1 bolded, and the components impacted by the change in gray (the reflexive-transitive closure of 1 in the inverse graph). Based on these results, we call *check* on 5, but not on 6.

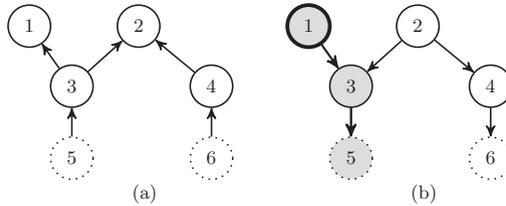


Fig. 1. Dependency graph where component 1 is changed, impacting 3 and 5.

3.3 Correctness Approach

For correctness, we intuitively show that executing only impacted and fresh vertices that are checkable is enough in the new revision, since the result of executing unimpacted vertices is the same as in the old revision. This means that if we have access to the results of checking vertices in the old revision, we can use those results to obtain the complete outcome for all checkable vertices in the new revision, without going through the work usually required.

Having constructed the set T of tuples of checkable vertices and outcomes from the impacted, fresh, and unimpacted vertices, we can ask (1) whether T is *complete*, i.e., whether it contains outcomes for all checkable vertices in V' , and (2) whether the outcomes in T are *sound*, i.e., if they are same as if we had explicitly called *check* on the associated vertices.

To be able to prove soundness and completeness, we need to assume several properties relating the dependency graphs and outcomes of executing vertices in both revisions. Informally, we make the following assumptions:

- A1: The direct dependencies of a vertex v are the same in both revisions if the artifact of v is the same in both revisions, i.e., if $f(v) = f'(v)$.
- A2: A vertex v with the same artifact in both revisions is checkable in the new revision iff v is checkable in the old revision.
- A3: The outcome of executing a checkable vertex v is the same in both revisions if the sets of vertices v depends on transitively are the same, and the artifact of each dependency is the same.

The last assumption implicitly rules out that the underlying operation (e.g., test execution) on a vertex is *nondeterministic*, which it can be in practice [41].

4 Model Encoding

In this section, we give an overview of our encoding in Coq of the formal model described in the previous section, using theories of finite sets and graphs from the MC library. We use a simplified version of Coq's specification language, Gallina.

4.1 Encoding in Coq

We represent the vertex set V' as a *finite type* (`finType`) V' , and its subset V as a *subtype* (`subType`) V , induced by a decidable predicate P on vertices in V' (of type `pred V'`). This allows us to define the graph g as a binary decidable relation g on V , i.e., a variable of type `rel V`, and use the MC library predicate `connect`

to express whether two vertices are transitively related in g . The inverse of g is defined as $[\text{rel } x y \mid g y x]$, which we write as g^{-1} . We use `connect` to form the set of vertices in the reflexive-transitive closure of a given vertex x with respect to a graph g , and a canonical big operator [7] to form the union of all such closures for elements in a given set m of modified vertices:

```
Def impacted (g : rel V) (m : {set V}) : {set V} :=
  \bigcup_(x | x \in m) [set y | connect g x y].
```

We characterize this function through MC's `reflect` ("if and only if"):

```
Thm impactedP g m x : reflect (∃ v, v \in m & connect g v x) (x \in impacted g m).
```

The MC library function `val` injects a subtype element into the corresponding supertype. We use this to capture impacted and fresh vertices in V' :

```
Def impacted_V' m : {set V'} := [set (val v) | v in impacted g^{-1} m].
Def fresh_V' : {set V'} := [set v | ¬ P v].
```

We represent the set of artifacts A as a type A with decidable equality (`eqType`), and functions f and f' as regular Coq functions `f` and `f'`. This allows us to define the set of modified vertices in V' , and then take the union (operator `∪`) of impacted and fresh vertices:

```
Def mod_V : {set V} := [set v | f v != f' (val v)].
Def impacted_fresh_V' : {set V'} := impacted_V' mod_V ∪ fresh_V'.
```

We then use a predicate `checkable` to form the subset of vertices in V' that can be executed:

```
Def chk_impacted_fresh_V' : {set V'} := [set v in impacted_fresh_V' | checkable v].
```

We use a function `check`, which takes a vertex and returns a term in a result type R (an `eqType`, e.g., `bool`), to define a sequence of vertices and results:

```
Def res_impacted_fresh_V' : seq (V' * R) :=
  [seq (v, check v) | v ← enum chk_impacted_fresh_V'].
```

Note that by using a sequence instead of a finite set for these tuples, we ensure R can be any type with decidable equality, such as a message of arbitrary length.

4.2 Correctness Statements

For stating and proving correctness, we assume we have dependency graphs for the old and new revision, as well as definitions of whether vertices are checkable, and checking functions:

```
Vars (g : rel V) (g' : rel V').
Vars (checkable : pred V) (checkable' : pred V') (check : V → R) (check' : V' → R).
```

We then define the graph g for vertices in V' , named `g_V'`:

```
Def insub_g (x y : V') : bool := match insub x, insub y with
  Some x', Some y' => g x' y' | _, _ => false end.
Def g_V' : rel V' := [rel x y | insub_g x y].
```

This allows us to formulate the assumption A1 from above:

Hyp $\text{fg_eq} : \forall (v : V), f\ v = f'\ (\text{val } v) \rightarrow \forall (v' : V'), \text{g_V}' (\text{val } v)\ v' = \text{g}' (\text{val } v)\ v'$.

The assumption A2 is equally straightforward to define:

Hyp $\text{chk_f} : \forall v, f\ v = f'\ (\text{val } v) \rightarrow \text{checkable } v = \text{checkable}' (\text{val } v)$.

Finally, the assumption A3, when formalized, establishes a relation between vertices in g and g' :

Hyp $\text{chk_V} : \forall (v : V), \text{checkable } v \rightarrow \text{checkable}' (\text{val } v) \rightarrow$
 $(\forall (v' : V'), \text{connect } \text{g_V}' (\text{val } v)\ v' = \text{connect } \text{g}' (\text{val } v)\ v') \rightarrow$
 $(\forall (v' : V'), \text{connect } \text{g_V}' (\text{val } v)\ (\text{val } v') \rightarrow$
 $f\ v' = f'\ (\text{val } v')) \rightarrow \text{check } v = \text{check}' (\text{val } v)$.

We now assume we are given a sequence of results for checkable vertices in the old revision, and that this sequence is sound, complete, and duplicate-free:

Var $\text{res_V} : \text{seq } (V * R)$.

Hyp $\text{res_VP} : \forall v\ r, \text{reflect } (\text{checkable } v \wedge \text{check } v = r) ((v,r) \setminus \text{in } \text{res_V})$.

Hyp $\text{res_v_uniq} : \text{uniq } [\text{seq } \text{vr}.1 \mid \text{vr} \leftarrow \text{res_V}]$.

We can then filter the sequence of old results to locate unimpacted vertices in the new revision:

Def $\text{res_unimpacted_V}' : \text{seq } (V' * R) := [\text{seq } (\text{val } \text{vr}.1, \text{vr}.2) \mid$
 $\text{vr} \leftarrow \text{res_V} \ \& \ \text{val } \text{vr}.1 \setminus \text{notin } \text{impacted_V}' \ \text{mod_V}]$.

This allows us to form a final sequence of vertex-result pairs:

Def $\text{res_V}' : \text{seq } (V' * R) := \text{res_impacted_fresh_V}' \ ++ \ \text{res_unimpacted_V}'$.

For sanity-checking, we prove the absence of duplicates:

Def $\text{chk_V}' : \text{seq } V' := [\text{seq } \text{vr}.1 \mid \text{vr} \leftarrow \text{res_V}']$.

Thm $\text{chk_V}'\ \text{uniq} : \text{uniq } \text{chk_V}'$.

We prove that the sequence contains all checkable vertices in V' (completeness):

Thm $\text{chk_V}'\ \text{complete } (v : V') : \text{checkable}'\ v \rightarrow v \setminus \text{in } \text{chk_V}'$.

Finally, we prove that the results in the sequence are consistent with explicitly calling check' on all vertices in V' (soundness):

Thm $\text{chk_V}'\ \text{sound } (v : V')\ (r : R) : (v, r) \setminus \text{in } \text{res_V}' \rightarrow \text{checkable}'\ v \wedge \text{check}'\ v = r$.

The formal proofs, which we elide here, mostly reduce to reasoning over the connect predicate and inductively on graph paths.

5 Component Hierarchies

Let V be a set of vertices representing *fine-grained* components (e.g., methods), with dependency graph g_{\perp} . Let U be a different set of vertices representing *coarse-grained* components (e.g., files), associated with a function $p : U \rightarrow 2^V$ that defines a *partition* of V . The partition indicates how components in U *encapsulate* components in V , and is associated with a graph g_{\top} of vertices in U that is

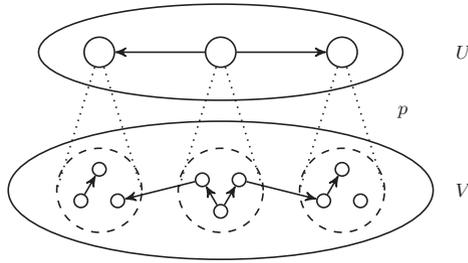


Fig. 2. Hierarchy with component sets U and V , partition p , and dependencies.

consistent with dependencies expressed in g_{\perp} . This approach can be repeated to produce component *hierarchies*, each time coalescing sets of finer-grained dependencies into single coarser-grained dependencies. Figure 2 illustrates a two-level hierarchy and its component dependencies.

Some change impact analysis techniques consider both fine-grained and coarse-grained component levels [11, 46, 60]. A key idea behind these techniques is to exploit the relationships between vertices across granularity levels. In particular, if a vertex $u \in U$ is unmodified after a change, we may be able to immediately conclude that all vertices $v \in p(u)$ are unmodified as well, potentially ruling out that a large subset of V is impacted. In this section, we formalize this intuition using our existing notions to express hierarchical change impact analysis.

5.1 Formal Model of Hierarchies

Let f_{\perp} and f'_{\perp} be the functions mapping vertices to artifacts for V and V' with $V \subseteq V'$, and let f_{\top} and f'_{\top} be the corresponding functions for U and U' with $U \subseteq U'$. Let p and p' be partition-inducing functions from U and U' to subsets of V and V' , respectively. We make the following assumptions:

H1: For all $u, u' \in U$ and $v, v' \in V$, if $u \neq u'$, $g_{\perp}(v, v')$, $v \in p(u)$, and $v' \in p(u')$, then $g_{\top}(u, u')$.

H2: For all $u \in U$, if $f_{\top}(u) = f'_{\top}(u)$, then $p(u) = p'(u)$.

H3: For all $u \in U$ and $v \in V$, if $f_{\top}(u) = f'_{\top}(u)$ and $v \in p(u)$, then $f_{\perp}(v) = f'_{\perp}(v)$.

Intuitively, H1 expresses that whenever two fine-grained components that reside in different coarse-grained components are related, there must be a corresponding relation between their respective coarse-grained components. H2 expresses that whenever a coarse-grained component is unchanged, it contains the same fine-grained components as before. Finally, H3 expresses that a fine-grained component is unchanged if the coarse-grained component that contains it is unchanged. Under these assumptions, there are essentially two distinct strategies we can use to leverage impact analysis for coarse-grained components to analyze fine-grained components.

Overapproximation strategy: Let U'_i be the set of impacted and fresh vertices in U' , computed as above without considering vertices in V' . Consider the set $V'_p = \bigcup_{u \in U'_i} p'(u)$ which contains fresh and *potentially* impacted vertices in V' .

Executing all checkable vertices in V'_p may perform needless work for unimpacted vertices, but completely elides analysis of g_\perp . This approach essentially corresponds to relying on comparing whole files to decide whether to rerun commands that operate on every component inside these files, as in `make`.

Compositional strategy: Let U_i be the set of impacted vertices in U , computed as above. Consider the set $V_p = \bigcup_{u \in U_i} p(u)$ of potentially impacted vertices in V . We use this set to scope further analysis. In particular, we use the subgraph g_p of g_\perp induced by V_p to precisely find the impacted vertices in V . While unimpacted vertices are then avoided, the additional analysis of g_p may be time-consuming to perform compared to the first strategy. At a high level, this strategy corresponds to the one used in RPS [11] and hybrid RTS [60].

5.2 Encoding and Correctness in Coq

To encode hierarchical analysis, we use finite types and functions (now suffixed by `top` and `bot`) in the same way as before, while adding partitioning assumptions:

Vars $(p : U \rightarrow \{\text{set } V\}) (p' : U' \rightarrow \{\text{set } V'\})$.

Hyp $p_pt : \text{partition } (\bigcup_{u \in U} [p\ u]) [set: V]$.

Hyp $p'_pt : \text{partition } (\bigcup_{u \in U'} [p'\ u]) [set: V']$.

For the overapproximation strategy, we first define impacted sets:

Def $if_top : \{\text{set } U'\} := \text{impacted_fresh_V}'\ f'_top\ f_top\ g_top$.

Def $p'_if_bot : \{\text{set } V'\} := \bigcup_{u \in U'} [p'\ u] \text{ in } if_top (p'\ u)$.

Under the assumptions outlined above, we then show formally that p'_if_bot is a superset of the results of analysis of v , V' , and the graph g_bot :

Thm $in_p' (v : V') : v \text{ in } \text{impacted_fresh_V}'\ f'_bot\ f_bot\ g_bot \rightarrow v \text{ in } p'_if_bot$.

The key fact we use to prove this theorem is the following:

Thm $connect_top_bot\ v\ v'\ u\ u' : v \text{ in } [p\ u] \rightarrow v' \text{ in } [p'\ u'] \rightarrow$
 $connect\ g_bot\ v\ v' \rightarrow connect\ g_top\ u\ u'$.

To encode the compositional strategy, we first define impacted sets:

Def $i_top : \{\text{set } U\} := \text{impacted } g_top^{-1} (\text{mod_V } f'_top\ f_top)$.

Def $p_i_bot : \{\text{set } V\} := \bigcup_{u \in U} [p\ u] \text{ in } i_top (p\ u)$.

Then, we define a subtype and accompanying graph:

Def $P_V_sub : \text{pred } V := \text{fun } v \Rightarrow v \text{ in } p_i_bot$.

Def $V_sub : \text{finType} := \text{sig_finType } P_V_sub$.

Def $g_bot_sub : \text{rel } V_sub := [\text{rel } x\ y \mid g_bot (val\ x) (val\ y)]$.

This allows us to use our previously defined analysis functions compositionally:

Def $\text{mod_V_sub} := [set\ v : V_sub \mid val\ v \text{ in } \text{mod_V } f'_bot\ f_bot]$.

Def $\text{impacted_V_sub} := \text{impacted } g_bot_sub^{-1} \text{ mod_V_sub}$.

Def $\text{impacted_V}'_sub := [set\ val (val\ v) \mid v \text{ in } \text{impacted_V_sub}]$.

Def $\text{impacted_fresh_V}'_sub := \text{impacted_V}'_sub \text{ :|: fresh_V}'\ P_bot$.

We finally show that the last set is the same as the one we would have obtained by directly analysing the graph g_bot :

```

Thm impacted_fresh_V'_sub_eq :
  impacted_fresh_V'_sub = impacted_fresh_V' f'_bot f_bot g_bot.

```

Using these definitions and results, we proved soundness and completeness for both strategies using the same approach as in Section 4.2.

6 Tool Implementation

While our core definitions of change impact analysis described in Section 4 are executable inside Coq, this does not mean they are efficient or that code extracted from the definitions is immediately usable. We describe two aspects of bringing verified Coq code into our tool CHIP: optimizations and encapsulation.

6.1 Optimizations

Our basic transitive closure function `impacted` is simple to reason about but not particularly fast in practice, since it fully explores the closures of *all* elements in the set of modified vertices. To mitigate this, we refined the function by leveraging the depth-first search function `dfs` from the `fingraph` MC module to incrementally compute the closure. `dfs` takes a graph as a function from vertices to neighbor sequences and a depth bound, and terminates as soon as it encounters a known vertex. We perform a stack-efficient left fold with `dfs` over an input sequence of vertices:

```

Def clos (l : seq V) : seq V := foldl (dfs g #|V|) [::] l.

```

Note that we set the `dfs` depth bound to the number of elements in the finite type `V` (written `#|V|`) to fully explore the graph `g`. However, one limitation of the MC `dfs` function is its linear-time sequence membership lookups. We therefore defined a better closure function with logarithmic membership lookup time using sets backed by red-black trees as found in the Coq standard library [2, 23]:

```

Fixpoint sdfs (g : V → seq V) (n : nat) (s : RBT.t) (x : V) : RBT.t :=
  if RBT.mem x s then s else
  if n is n'+1 then foldl (sdfs g n') (RBT.add x s) (g x) else s.
Def sclos (l : seq V) : seq V := RBT.elements (foldl (sdfs g #|V|) RBT.empty l).

```

We used this closure function to define a function `seq_impacted_fresh` which we proved extensionally equivalent to `impacted_fresh_V'` defined in Section 4.1. We also added many custom extraction directives in Coq to ensure the extracted code uses efficient OCaml library functions, e.g., for list operations [22].

6.2 Encapsulation

Before extraction to OCaml, we instantiate the finite types for graph vertices to *ordinal* finite types, which intuitively contain all natural numbers from 0 up to (but not including) some bound k . These numbers can then become machine integers during extraction, which allows us to provide a simple OCaml interface:

```

val impacted_fresh : int -> int -> (int -> string) ->
  (int -> string) -> (int -> int list) -> int list

```

Here, the first argument is the number of vertices in the new graph, while the second is the number of vertices in the old graph. After these integers follow two functions that map new and old vertices, respectively, to their artifacts in the form of OCaml strings. Then comes a function that defines the adjacent vertices of vertices in the old graph. The result is a list of impacted and fresh vertices.

Not all computationally meaningful types in Coq can be directly represented in OCaml's type system. Some function calls must therefore *circumvent* the type system by using calls to the special `Obj.magic` function [40]. We use this approach in our implementation of the above interface:

```
let impacted_fresh num_new num_old f' f succs =
  Obj.magic (ordinal_seq_impacted_fresh num_new num_old
    (Obj.magic (fun x -> char_list_of_string (f' x)))
    (Obj.magic (fun x -> char_list_of_string (f x)))
    (Obj.magic succs))
```

The interface and implementation for two-level compositional hierarchical selection is a straightforward extension, with an additional argument `p` of type `int -> int list` for between-level partitioning.

7 Evaluation of the Model

To evaluate our model and its Coq encoding, we performed an empirical study by integrating CHIP with a recently developed RTS tool, EKSTAZI, one RPS tool, iCOQ, and one build system, TUP. We then ran the modified RTS tool on open-source Java projects used to evaluate RTS techniques [26,37], the modified RPS tool on Coq projects used in its evaluation [11], and the modified build system on C/C++ projects. Finally, we compared the outcomes and running times with those for the unmodified versions of EKSTAZI, iCOQ, and TUP.

7.1 Tool Integration

Integrating CHIP with EKSTAZI was challenging, since EKSTAZI collects dependencies dynamically and builds only a flat list of dependencies rather than an explicit graph. To overcome this limitation, we modified EKSTAZI to build an explicit graph by maintaining a mapping from method callers to their callees. The integration with iCOQ was also challenging because of the need for hierarchical selection of proofs and support for deletion of dependency graph vertices. We handle deletion of a vertex in iCOQ by temporarily adding it to the new graph with a different artifact (checksum) from before, marked as non-checkable; then, after selection, we purge the vertex. In contrast, the integration with TUP was straightforward, since TUP stores dependencies in an SQLite database. We simply query this database to obtain a graph in the format expected by CHIP.

7.2 Projects

RTS: We use 10 GitHub projects. Table 1 (top) shows the name of each project, the number of lines of code (LOC) and the number of tests in the latest version control revision we used in our experiments, the SHA of the latest revision, and

Table 1. List of Projects Used in the Evaluation (RTS at the Top, RPS in the Middle, and TUP at the Bottom).

Project	LOC	#Tests	SHA	URL (github.com/)
Asterisk	57,219	257	e36c655f	asterisk-java/asterisk-java
Codec	22,009	887	58860269	apache/commons-codec
Collections	66,356	24,589	d83572be	apache/commons-collections
Lang	81,533	4,119	c3de2d69	apache/commons-lang
Math	186,388	4,858	eb57d6d4	apache/commons-math
GraphHopper	70,615	1,544	14d2d670	graphhopper/graphhopper
La4j	13,823	799	e77dca70	vkostyukov/la4j
Planner	82,633	398	f12e8600	opentripplanner/OpenTripPlanner
Retrofit	20,476	603	7a0251cc	square/retrofit
Truth	29,591	1,448	14f72f73	google/truth
Total	630,643	39,502	N/A	N/A
Project	LOC	#Proofs	SHA	URL
Flocq	33,544	943	4161c990	gitlab.inria.fr/flocq/flocq
StructTact	2,497	187	8f1bc10a	github.com/uwplse/StructTact
UniMath	45,638	755	5e525f08	github.com/UniMath/UniMath
Verdi	57,181	2,784	15be6f61	github.com/uwplse/Verdi
Total	138,860	4,669	N/A	N/A
Project	LOC	#Cmds	SHA	URL (github.com/)
guardcheader	656	5	dbd1c0f	kalrish/guardcheader
LazyIterator	1,276	18	d5f0b64	matthiasvegh/LazyIterator
libhash	347	10	b22c27e	fourier/libhash
Redis	162,366	213	39c70e7	antirez/redis
Shaman	925	7	73c048d	HalosGhost/shaman
Tup	200,135	86	f77dbd4	gittup/tup
Total	365,705	339	N/A	N/A

URL on GitHub. We chose these projects because they are popular Java projects (in terms of stars) on GitHub, use the Maven build system (supported by EKSTAZI), and were recently used in RTS research [37, 60].

RPS: We use 4 Coq projects. Table 1 (middle) shows the name of each project, the number of LOC and the number of proofs in the latest revision we used, the latest revision SHA, and URL. We chose these projects because they were used in the evaluation of iCOQ [11]; as in that evaluation, we used 10 revisions of StructTact and 24 revisions of the other projects.

Build system: We use 6 GitHub projects. Table 1 (bottom) shows the name of each project, the number of LOC and the number of build commands in the latest revision we used, the latest revision SHA, and URL. We chose these projects from the limited set of projects on GitHub that use TUP. We looked for projects that could be built successfully and had at least five revisions; the largest project that met these requirements, in terms of LOC, was TUP itself.

7.3 Experimental Setup

Our experimental setup closely follows recent work on RTS [37, 60]. That is, our scripts (1) clone one of the projects; (2) integrate the (modified) EKSTAZI, iCOQ, or TUP; and (3) execute tests on, check proofs for, or build the (up to) 24 latest revisions. For each run, we recorded the end-to-end execution time, which includes time for the entire build run. We also recorded the execution time for change impact analysis alone. Finally, we recorded the number of executed tests,

Table 2. Execution Time and CIA Time in Seconds for EKSTAZI and CHIP.

Project	Total			CIA	
	RetestAll	Ekstazi	Chip	Ekstazi	Chip
Asterisk	461.92	188.67	194.65	2.74	6.51
Codec	896.00	135.11	136.35	2.44	4.10
Collections	2,754.99	342.07	350.95	2.87	9.31
Lang	1,844.19	359.36	367.16	2.71	8.68
Math	2,578.09	1,459.98	1,495.71	1.79	7.13
GraphHopper	1,871.01	423.63	449.94	11.19	21.33
La4j	272.96	202.10	209.41	1.12	3.91
Planner	4,811.63	1,144.09	1,228.61	40.62	89.17
Retrofit	1,181.09	722.14	747.76	11.30	19.97
Truth	745.11	700.26	724.22	3.03	8.82
Total	17,416.99	5,677.41	5,904.76	79.81	178.93

Table 3. Execution/CIA Time in Seconds for iCOQ and CHIP.

Project	Total			CIA	
	RecheckAll	iCoq	Chip	iCoq	Chip
Flocq	1,028.01	313.08	318.19	50.65	53.43
StructTact	45.86	43.90	44.49	14.45	14.98
UniMath	14,989.09	1,910.56	2,026.75	124.79	239.12
Verdi	37,792.07	3,604.23	4,637.27	139.09	1,171.57
Total	53,855.03	5,871.76	7,026.70	328.98	1,479.10

proofs, or commands, which we use to verify the correctness of the results, i.e., we checked that the results for the unmodified tool and CHIP were equivalent. We ran all experiments on a 4-core Intel i7-6700 CPU @ 3.40GHz machine with 16GB of RAM, running Ubuntu Linux 17.04. We confirmed that the execution time for each experiment was similar across several runs.

7.4 Results

RTS: Table 2 shows the execution times for EKSTAZI. Column 1 shows the names of the projects. Columns 2 to 4 show the cumulative end-to-end time for RetestAll (i.e., running all tests at each revision), the unmodified RTS tool, and the RTS tool with CHIP. Columns 5 and 6 show the cumulative time for change impact analysis (CIA time). The last row in the table shows the cumulative execution time across all projects. We have several findings. First, EKSTAZI with CHIP performs significantly better than RetestAll, and only slightly worse than the unmodified tool. Considering that we did not prioritize optimizing the integration, we believe that the current execution time differences are small. Second, the CIA time using CHIP is slightly higher than the CIA time for the unmodified tool, but we believe this could be addressed by integrating CHIP via the Java Native Interface (JNI). The selected tests for all projects and revisions were the same for the unmodified EKSTAZI and EKSTAZI with CHIP.

RPS: Table 3 shows the total proof checking time for iCOQ and the CIA time for iCOQ and CHIP. All time values are cumulative time across all the revisions we used. We find that iCOQ with CHIP has only marginal differences in performance from iCOQ for all but the largest project, Verdi. While iCOQ with

CHIP is notably slower in that case, it still saves a significant fraction of time from checking every revision from scratch (RecheckAll). StructTact is an outlier in that RecheckAll is actually faster than both iCOQ and iCOQ with CHIP, due to the overhead from bookkeeping and graph processing in comparison to the project’s relatively small size. The selected proofs for all projects and revisions were the same for the unmodified iCOQ and iCOQ with CHIP.

Build system: Table 4 shows the total execution time for TUP and the CIA time for TUP and CHIP. All time values are cumulative time across all the revisions we used. Unfortunately, the build time for most of the projects is short. However, we can still observe that CHIP takes only slightly more time than the original tool to perform change impact analysis. In the future, we plan to evaluate our toolchain on larger projects. The lists of commands for all projects and all revisions were the same for the unmodified TUP and TUP with CHIP.

Overall, we believe these results indicate that our formal model is practically relevant and that it is feasible to use CHIP as a verified component for change impact analysis in real-world tools.

8 Related Work

Formalizations of graph algorithms: Pottier [49] encoded and verified Kosaraju’s algorithm for computing strongly connected graph components in Coq. He also derived a practical program for depth-first search by extracting Coq code to OCaml, demonstrating the feasibility of extraction for graph-based programs. Théry subsequently formalized a similar encoding of Kosaraju’s algorithm in Coq using the MC `fingraph` module [56]. Théry and Cohen then formalized and proved correct Tarjan’s algorithm for computing strongly connected graph components in Coq [13, 16]. Our formalization takes inspiration from Théry and Cohen’s work, and adapts some of their definitions and results in a more applied context, with focus on performance of extracted code. Similar graph algorithm formalizations have also been done in the Isabelle/HOL proof assistant [35]. In work particularly relevant to build systems, Guéneau et al. [32] verified both the functional correctness and time complexity of an incremental graph cycle detection algorithm in Coq. In contrast to our reasoning on pure functions and use of extraction, they reason directly on imperative OCaml code.

Formalizations of build systems: Christakis et al. [15] formalized a general build language called CloudMake in the Dafny verification tool. Their language is a purely functional subset of JavaScript, and allows describing dependencies between executions of tools and files. Having embedded their language in Dafny, they verify that builds with cached files are equivalent to builds from scratch. In contrast to the focus on generating files in CloudMake, we consider a formal model with an explicit dependency graph and an operation *check* on vertices whose output is not used as input to other operations. The CloudMake formalization assumes an arbitrary operation *exec* that can be instantiated using

Table 4. Execution Time in Milliseconds for TUP and CHIP.

Project	CIA		
	Tup	Tup	Chip
guardheader	20,358	1,788	1,785
LazyIterator	61,476	869	1,007
libhash	15,279	433	446
Redis	68,076	1,919	4,779
Shaman	8,702	609	614
Tup	87,547	1,949	4,168
Total	261,438	7,567	12,799

Dafny’s module refinement system; we use Coq section variables to achieve similar parametrization for *check*. We view our Coq development as a *library* useful to tool builders, rather than a separate language that imposes a specific idiom for expressing dependencies and build operations.

Mokhov et al. [45] presented an analysis of several build systems, including a definition what it means for such systems to be correct. Their correctness formulation is similar to that of Christakis et al. for cached builds, and relies on a notion of abstract persistent stores expressed via monads. Our vertices and artifacts correspond quite closely to their notions of *keys* and *values*, respectively. However, their basic concepts are given as Haskell code, which has less clear meaning and a larger trusted base than Coq or Dafny code. Moreover, they provide no formal proofs. Mokhov et al. [44] subsequently formalized in Haskell a static analysis of build dependencies as used in the Dune build system.

Stores could be added to our model, e.g., by letting checkable vertices be associated with commands that take lists of file names and the current store state as parameters, producing a new state. However, this would in effect entail defining a specific build language inside Coq, which we consider outside the scope of our library and tool.

9 Conclusion

We presented a formalization of change impact analysis and its encoding and correctness proofs in the Coq proof assistant. Our formal model uses finite sets and graphs to capture system components and their interdependencies before and after a change to a system. We locate impacted vertices that represent, e.g., tests to be run or build commands to be executed, by computing transitive closures in the pre-change dependency graph. We also considered two strategies for change impact analysis of hierarchical systems of components. We extracted optimized impact analysis functions in Coq to executable OCaml code, yielding a verified tool dubbed CHIP. We then integrated CHIP with a regression test selection tool for Java, EKSTAZI, one regression proof selection tool for Coq itself, iCOQ, and one build system, TUP, by replacing their existing components for impact analysis. We evaluated the resulting toolchains on several open-source projects by comparing the outcome and running time to those for the respective original tools. Our results show the same outcomes with only small differences in running time, corroborating the adequacy of our model and the feasibility of practical verified tools for impact analysis. We also believe our Coq library can be used as a basis for proving correct domain-specific incremental techniques that rely on change impact analysis, e.g., regression test selection for Java and regression proof selection for type theories.

Acknowledgments

The authors thank Ben Buhse, Cyril Cohen, Pengyu Nie, Zachary Tatlock, Thomas Wei, Chenguang Zhu, and the anonymous reviewers for their comments and feedback on this work. This work was partially supported by the US National Science Foundation under Grant No. CCF-1652517.

References

1. Acharya, M., Robinson, B.: Practical change impact analysis based on static program slicing for industrial software systems. In: International Conference on Software Engineering. pp. 746–755. ACM, New York, NY, USA (2011). <https://doi.org/10.1145/1985793.1985898>
2. Appel, A.W.: Efficient verified red-black trees (2011), <https://www.cs.princeton.edu/~appel/papers/redblack.pdf>, last accessed 21 Feb 2020.
3. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society, Los Alamitos, CA, USA (1996)
4. Arnold, R.S., Bohner, S.A.: Impact analysis - towards a framework for comparison. In: International Conference on Software Maintenance. pp. 292–301. IEEE Computer Society, Washington, DC, USA (1993). <https://doi.org/10.1109/ICSM.1993.366933>
5. Bazel team: Bazel Blog, <https://blog.bazel.build>, last accessed 20 Feb 2020.
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: the calculus of inductive constructions. Springer, Heidelberg, Germany (2004). <https://doi.org/10.1007/978-3-662-07964-5>
7. Bertot, Y., Gonthier, G., Ould Biha, S., Pasca, I.: Canonical big operators. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) International Conference on Theorem Proving in Higher Order Logics. LNCS, vol. 5170, pp. 86–101. Springer, Heidelberg, Germany (2008). https://doi.org/10.1007/978-3-540-71067-7_11
8. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In: SIGCOMM Conference. pp. 265–276. ACM, New York, NY, USA (2005). <https://doi.org/10.1145/1080091.1080123>
9. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Abadi, M., Ito, T. (eds.) Theoretical Aspects of Computer Software. LNCS, vol. 1281, pp. 515–529. Springer, Heidelberg, Germany (1997). <https://doi.org/10.1007/BFb0014565>
10. Cai, H., Santelices, R.: A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *J. Syst. Softw.* **103**(C), 248–265 (2015). <https://doi.org/10.1016/j.jss.2015.02.018>
11. Celik, A., Palmkog, K., Gligoric, M.: iCoq: Regression proof selection for large-scale verification projects. In: International Conference on Automated Software Engineering. pp. 171–182. IEEE Computer Society, Washington, DC, USA (2017). <https://doi.org/10.1109/ASE.2017.8115630>
12. Celik, A., Palmkog, K., Gligoric, M.: A regression proof selection tool for Coq. In: International Conference on Software Engineering, Tool Demonstrations. pp. 117–120. ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3183440.3183493>
13. Chen, R., Cohen, C., Lévy, J.J., Merz, S., Théry, L.: Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) International Conference on Interactive Theorem Proving. pp. 13:1–13:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). <https://doi.org/10.4230/LIPIcs.ITP.2019.13>
14. Chodorow, K.: Trimming the (build) tree with Bazel, <https://www.kchodorow.com/blog/2015/07/23/trimming-the-build-tree-with-bazel/>, last accessed 20 Feb 2020.
15. Christakis, M., Leino, K.R.M., Schulte, W.: Formalizing and verifying a modern build language. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) Symposium on For-

- mal Methods. LNCS, vol. 8442, pp. 643–657. Springer, Cham, Switzerland (2014). https://doi.org/10.1007/978-3-319-06410-9_43
16. Cohen, C., Théry, L.: Formalization of Tarjan 72 algorithm in Coq with Mathematical Components and SSReflect, <https://github.com/CohenCyril/tarjan>, last accessed 21 Feb 2020.
 17. Coquand, T., Huet, G.: The calculus of constructions. *Information and Computation* **76**(2), 95–120 (1988). [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3)
 18. Coquand, T., Paulin-Mohrin, C.: Inductively defined types. In: Martin-Löf, P., Mints, G. (eds.) *International Conference on Computer Logic*. LNCS, vol. 417, pp. 50–66. Springer, Heidelberg, Germany (1990). https://doi.org/10.1007/3-540-52335-9_47
 19. Cruz-Filipe, L., Letouzey, P.: A large-scale experiment in executing extracted programs. *Electronic Notes in Theoretical Computer Science* **151**(1), 75–91 (2006). <https://doi.org/10.1016/j.entcs.2005.11.024>
 20. Delaware, B., Suriyakarn, S., Pit-Claudel, C., Ye, Q., Chlipala, A.: Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.* **3**(ICFP) (2019). <https://doi.org/10.1145/3341686>
 21. Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., Kandula, S.: CloudBuild: Microsoft’s distributed and caching build service. In: *International Conference on Software Engineering, Software Engineering in Practice*. pp. 11–20. ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2889160.2889222>
 22. ExtLib team: OCaml Extended standard Library, <https://github.com/ygrek/ocaml-extlib>, last accessed 20 Feb 2020.
 23. Filliâtre, J.C., Letouzey, P.: Functors for proofs and programs. In: Schmidt, D. (ed.) *European Symposium on Programming*. LNCS, vol. 2986, pp. 370–384. Springer, Heidelberg, Germany (2004). https://doi.org/10.1007/978-3-540-24725-8_26
 24. Fonseca, P., Zhang, K., Wang, X., Krishnamurthy, A.: An empirical study on the correctness of formally verified distributed systems. In: *European Conference on Computer Systems*. pp. 328–343. ACM, New York, NY, USA (2017). <https://doi.org/10.1145/3064176.3064183>
 25. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *International Conference on Theorem Proving in Higher Order Logics*. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg, Germany (2009). https://doi.org/10.1007/978-3-642-03359-9_23
 26. Gligoric, M., Eloussi, L., Marinov, D.: Practical regression test selection with dynamic file dependencies. In: *International Symposium on Software Testing and Analysis*. pp. 211–222. ACM, New York, NY, USA (2015). <https://doi.org/10.1145/2771783.2771784>
 27. Gligoric, M., Schulte, W., Prasad, C., van Velzen, D., Narasamya, I., Livshits, B.: Automated migration of build scripts using dynamic analysis and search-based refactoring. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. pp. 599–616. ACM, New York, NY, USA (2014). <https://doi.org/10.1145/2714064.2660239>
 28. Gonthier, G.: Formal proof—the four-color theorem. *Notices of the American Mathematical Society* **55**(11), 1382–1393 (2008), <http://www.ams.org/notices/200811/tx081101382p.pdf>
 29. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O’Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev,

- A., Tassi, E., Théry, L.: A machine-checked proof of the odd order theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) International Conference on Interactive Theorem Proving. LNCS, vol. 7998, pp. 163–179. Springer, Heidelberg, Germany (2013). https://doi.org/10.1007/978-3-642-39634-2_14
30. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* **3**(2), 95–152 (2010). <https://doi.org/10.6092/issn.1972-5787/1979>
 31. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. In: International Conference on Functional Programming. pp. 163–175. ACM, New York, NY, USA (2011). <https://doi.org/10.1145/2034773.2034798>
 32. Guéneau, A., Jourdan, J.H., Charguéraud, A., Pottier, F.: Formal proof and analysis of an incremental cycle detection algorithm. In: Harrison, J., O’Leary, J., Tolmach, A. (eds.) International Conference on Interactive Theorem Proving. pp. 18:1–18:20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). <https://doi.org/10.4230/LIPIcs.ITP.2019.18>
 33. Kell, S., Mulligan, D.P., Sewell, P.: The missing link: Explaining ELF static linking, semantically. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 607–623. ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2983990.2983996>
 34. Lahiri, S.K., Vaswani, K., Hoare, C.A.R.: Differential static analysis: Opportunities, applications, and challenges. In: Workshop on Future of Software Engineering Research. pp. 201–204. ACM, New York, NY, USA (2010). <https://doi.org/10.1145/1882362.1882405>
 35. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: Conference on Certified Programs and Proofs. pp. 137–146. ACM, New York, NY, USA (2015). <https://doi.org/10.1145/2676724.2693165>
 36. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: International Conference on Software Engineering. pp. 308–318. IEEE Computer Society, Washington, DC, USA (2003). <https://doi.org/10.1109/ICSE.2003.1201210>
 37. Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., Marinov, D.: An extensive study of static regression test selection in modern software evolution. In: International Symposium on Foundations of Software Engineering. pp. 583–594. ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2950290.2950361>
 38. Lehnert, S.: A review of software change impact analysis. Tech. rep., Technische Universität Ilmenau, Ilmenau, Germany (2011), <https://nbn-resolving.org/urn:nbn:de:gbv:ilm1-2011200618>
 39. Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* **52**(7), 107–115 (2009). <https://doi.org/10.1145/1538788.1538814>
 40. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) Types for Proofs and Programs. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg, Germany (2003). https://doi.org/10.1007/3-540-39185-1_12
 41. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: International Symposium on Foundations of Software Engineering. pp. 643–653. ACM, New York, NY, USA (2014). <https://doi.org/10.1145/2635868.2635920>
 42. MathComp team: Mathematical Components project, <https://math-comp.github.io>, last accessed 20 Feb 2020.
 43. Mitchell, N.: Shake before building: Replacing Make with Haskell. In: International Conference on Functional Programming. pp. 55–66. ACM, New York, NY, USA (2012). <https://doi.org/10.1145/2364527.2364538>

44. Mokhov, A., Lukyanov, G., Marlow, S., Dimino, J.: Selective applicative functors. *Proc. ACM Program. Lang.* **3**(ICFP), 90:1–90:29 (2019). <https://doi.org/10.1145/3341694>
45. Mokhov, A., Mitchell, N., Peyton Jones, S.: Build systems à la carte. *Proc. ACM Program. Lang.* **2**(ICFP), 79:1–79:29 (2018). <https://doi.org/10.1145/3236774>
46. Orso, A., Shi, N., Harrold, M.J.: Scaling regression testing to large software systems. In: *International Symposium on Foundations of Software Engineering*. pp. 241–251. ACM, New York, NY, USA (2004). <https://doi.org/10.1145/1041685.1029928>
47. Palmiskog, K., Celik, A., Gligoric, M.: Chip code release 1.0, <https://github.com/palmiskog/chip/releases/tag/v1.0>, last accessed 20 Feb 2020.
48. Pollock, L.L., Soffa, M.L.: Incremental compilation of optimized code. In: *Symposium on Principles of Programming Languages*. pp. 152–164. ACM, New York, NY, USA (1985). <https://doi.org/10.1145/318593.318629>
49. Pottier, F.: Depth-first search and strong connectivity in Coq. In: Baelde, D., Alglave, J. (eds.) *Journées francophones des langages applicatifs (JFLA)*. Le Val d’Ajol, France (2015), <https://hal.inria.fr/hal-01096354>
50. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: A tool for change impact analysis of Java programs. In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. pp. 432–448. ACM, New York, NY, USA (2004). <https://doi.org/10.1145/1028976.1029012>
51. Rothermel, G.: Efficient, Effective Regression Testing Using Safe Test Selection Techniques. Ph.D. thesis, Clemson University, Clemson, SC, USA (1996)
52. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. *Transactions on Software Engineering and Methodology* **6**(2), 173–210 (1997). <https://doi.org/10.1145/248233.248262>
53. Rungta, N., Person, S., Branchaud, J.: A change impact analysis to characterize evolving program behaviors. In: *International Conference on Software Maintenance*. pp. 109–118. IEEE Computer Society, Washington, DC, USA (2012). <https://doi.org/10.1109/ICSM.2012.6405261>
54. Shal, M.: Build system rules and algorithms (2009), http://gittup.org/tup/build_system_rules_and_algorithms.pdf, last accessed 21 Feb 2020.
55. Skoglund, M., Runeson, P.: Improving class firewall regression test selection by removing the class firewall. *International Journal of Software Engineering and Knowledge Engineering* **17**(3), 359–378 (2007). <https://doi.org/10.1142/S0218194007003306>
56. Théry, L.: Formally-Proven Kosaraju’s algorithm (2015), <https://hal.archives-ouvertes.fr/hal-01095533>, last accessed 21 Feb 2020.
57. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the Raft consensus protocol. In: *Conference on Certified Programs and Proofs*. pp. 154–165. ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2854065.2854081>
58. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: *Conference on Programming Language Design and Implementation*. pp. 283–294. ACM, New York, NY, USA (2011). <https://doi.org/10.1145/1993498.1993532>
59. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability* **22**(2), 67–120 (2012). <https://doi.org/10.1002/stvr.430>

60. Zhang, L.: Hybrid regression test selection. In: International Conference on Software Engineering. pp. 199–209. ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3180155.3180198>
61. Zhang, L., Kim, M., Khurshid, S.: FaultTracer: a spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process* **25**, 1357–1383 (2013). <https://doi.org/10.1002/smr.1634>

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

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





What's Decidable About Program Verification Modulo Axioms?*

Umang Mathur[✉], P. Madhusudan, and Mahesh Viswanathan

University of Illinois, Urbana Champaign, USA

Abstract. We consider the decidability of the verification problem of programs *modulo axioms* — automatically verifying whether programs satisfy their assertions, when the function and relation symbols are interpreted as arbitrary functions and relations that satisfy a set of first-order axioms. Though verification of uninterpreted programs (with no axioms) is already undecidable, a recent work introduced a subclass of *coherent* uninterpreted programs, and showed that they admit decidable verification [26]. We undertake a systematic study of various natural axioms for relations and functions, and study the decidability of the coherent verification problem. Axioms include relations being reflexive, symmetric, transitive, or total order relations, functions restricted to being associative, idempotent or commutative, and combinations of such axioms as well. Our comprehensive results unearth a rich landscape that shows that though several axiom classes admit decidability for coherent programs, coherence is not a panacea as several others continue to be undecidable.

1 Introduction

Programs are proved correct against safety specifications typically by induction—the induction hypothesis is specified using *inductive invariants* of the program, and one proves that the reachable states of the program stays within the region defined by the invariants, inductively. Though there has been tremendous progress in the field of *decidable logics* for proving that invariants are inductive, finding inductive invariants is almost never fully automatic. And completely automated verification of programs is almost always undecidable.

Programs can be viewed as working over a data-domain, with variables storing values over this domain and being updated using constants, functions and relations defined over that domain. Apart from the notable exception of finite data domains, program verification is typically undecidable when the data domain is infinite. In a recent paper, Mathur et. al. [26] establish new decidability results when the data domain is infinite. Two crucial restrictions are imposed — data domain functions and relations are assumed to be *uninterpreted* and programs are assumed to be *coherent* (the meaning of coherence is discussed later

* Umang Mathur is partially supported by a Google PhD Fellowship. P. Madhusudan is partially supported by NSF CCF 1527395. Mahesh Viswanathan is partially supported by NSF CCF 1901069

in this introduction). The theory of uninterpreted functions is an important theory in SMT solvers that is often used (in conjunction with other theories) to solve feasibility of loop-free program snippets, in bounded model-checking, and to validate verification conditions. The salient aspect of [26] is to show that entire program verification is decidable for the class of coherent programs, without any user-provided inductive invariants (like loop invariants). While the results of [26] were mainly theoretical, there has been recent work on applying this theory to verifying memory-safety of heap-manipulating programs [28].

Data domain functions and relations used in a program usually satisfy special properties and are not, of course, entirely uninterpreted. The results of [26] can be seen as an approximate/abstraction-based verification method in practice — if the program verifies assuming functions and relations to be uninterpreted, then the program is correct for *any* data domain. However, properties of the data domain are often critical in establishing correctness. For example, in order to prove that a sorting program results in sorted arrays, it is important that the binary relation $<$ used to compare elements of the array is a total ordering on the underlying data sort. Consequently, constraining the data domain to satisfy certain axioms results in more accurate modeling for verification.

In this paper, we undertake a systematic study of the verification of uninterpreted programs when the data-domains are constrained using theories specified by (universally quantified) axioms. The choice of the axioms we study are guided by two principles. First, we study natural mathematical properties of functions and relations. Second, we choose to study axioms that have a decidable *quantifier-free* fragment of first order logic. The reason is that even single program executions (as defined in Section 3.2) can easily encode quantifier-free formulae (by computing the terms in variables, and assert Boolean combinations of atomic relations and equality on them). Since we are seeking decidable verification for programs *with loops/iteration*, it makes little sense to examine axioms where even verification of single executions is undecidable.

Coherence modulo theories: Mathur et. al. [26] define a subclass of programs, called *coherent programs*, for which program verification on uninterpreted domains is decidable; without the restriction of *coherence*, program verification on uninterpreted domains is undecidable. Since our framework is strictly more powerful, we adapt the notion of coherence to incorporate theories. A coherent program [26] is one where all executions satisfy two properties — memoizing and early-assumes. The memoizing property demands that the program computes any term, modulo congruence induced by the equality assumes in the execution, only once. More precisely, if an execution recomputes a term, the term should be stored in a current variable. The early-assumes restriction demands, intuitively, that whenever the program assumes two terms to be equal, it should do so early, before computing superterms of them.

We adapt the above notion to *coherence modulo theories*¹. The memoizing and early-assumes property are now required modulo the equalities that are entailed by the axioms. More precisely, if the theory is characterized by a set of axioms \mathcal{A} , the memoizing property demands that if a program computes a term t and there was another term t' that it had computed earlier which is equivalent to t modulo the assumptions made thus far *and the axioms* \mathcal{A} , then t' must be currently stored in a variable. Similarly, the early-assumes condition is also with respect to the axioms — if the program execution observes a new assumption of equality or a relation holding between terms, then we require that any equality entailed newly by it, the previous assumptions *and the axioms* \mathcal{A} do not involve a dropped term. This is a smooth extension of the notion of coherence from [26]; when $\mathcal{A} = \emptyset$, we essentially retrieve the notion from [26].

Main Contributions

Our first contribution is an extension of the notion of coherence in [26] to handle the presence of axioms, as described above; this is technically nontrivial and we provide a natural extension.

Under the new notion of coherence, we first study axioms on relations. The EPR (effectively propositional reasoning) [37] fragment of first order logic is one of the few fragments of first order logic that is decidable, and has been exploited for bounded model-checking and verification condition validation in the literature [34,33,32]. We study axioms written in EPR (i.e., universally quantified formulas involving only relations) and show that verification for even coherent programs, modulo EPR axioms, is undecidable.

Given the negative result on EPR, we look at particular natural axioms for relations, which are nevertheless expressible in EPR. In particular, we look at reflexivity, irreflexivity, and symmetry axioms, and show that verification of coherent programs is decidable when the interpretation of some relational symbols is constrained to satisfy these axioms. Our proof proceeds by instrumenting the program with auxiliary `assume` statements that preserve coherence and subtle arguments that show that verification can be reduced to the case without axioms; decidability then follows from results established in [26].

We then show a much more nontrivial result that verification of coherent programs remains decidable when some relational symbols are constrained to be transitive. The proof relies on new automata constructions that compute streaming congruence closures while interpreting the relations to be transitive.

Furthermore, we show that combinations of reflexivity, irreflexivity, symmetry, and transitivity, admit a decidable verification problem for coherent program. Using this observation, we conclude decidability of verification when certain relations are required to be strict partial orders (irreflexive and transitive) or equivalence relations.

¹ We adapt the definition in a way that preserves the spirit of the definition of coherence. Moreover, if we do not adapt the definition, essentially all axioms classes we study in this paper would be undecidable.

We then consider axioms that capture total orders and show that they too admit a decidable coherent verification problem. Total orders are also expressible in EPR and their formulation in EPR has been used in program verification, as they can be used in lieu of the ordering on integers when only ordering is important. For example, they can be used to model data in sorting algorithms, array indices in modeling distributed systems to model process ids and the states of processes, etc. [34,33].

Our next set of results consider axioms on functions. Associativity and commutativity are natural and fundamental properties of functions (like $+$ and $*$) and are hence natural ways to capture/abstract using these axioms. (See [14] where such abstractions are used in program analysis.) We first show that verification of coherent programs is decidable when some functions are assumed to be commutative or idempotent. Our proof, similar to the case of reflexive and symmetric relations, relies on reducing verification to the case without axioms using program instrumentation that capture the commutativity and idempotence axioms. However, when a function is required to be associative, the verification problem for coherent programs becomes undecidable. This undecidability result was surprising to us.

The decidability results established for properties of individual relation or function symbols discussed above can be combined to yield decidable verification modulo a set of axioms. That is, the verification of coherent programs with respect to models where relational symbols satisfy some subset of reflexivity/irreflexivity/symmetry/transitivity axioms or none, and function symbols are either uninterpreted, commutative, or idempotent, is decidable.

Decidability results outlined above, apply to programs that are coherent modulo the axioms/theories. However, given a program, in order to verify it using our techniques, we would also like to decide whether the program *is* coherent modulo axioms. We prove that for all the decidable axioms above, checking whether programs are coherent modulo the axioms is a decidable problem. Consequently, under these axioms, we can both check whether programs are coherent modulo the axioms and if they are, verify them.

There are several other results that we mention only in passing. For instance, we show that even for single executions, verifying them modulo equational axioms is undecidable as it is closely related to the word problem for groups. And our positive results for program verification under axioms for functions (commutativity, idempotence), also shows that bounded model-checking under such axioms is decidable, which can have its own applications.

Due to the large number of results and technically involved proofs, we give only the main theorems and proof gists for some of these in the paper; details can be found in [27].

2 Illustrative Example

Consider the problem of searching for an element k in a sorted list. There are two simple algorithms for this problem. Algorithm 1 walks through the list from

```

assume (T ≠ F);
found := F;
stop := F;
exists := F;
sorted := T;
while( x ≠ NIL) {
  if( stop = F) then {
    if( k = key(x)) then found := T;
    if( k ≤ key(x)) then stop := T;
  }
  if( k = key(x)) then exists := T;
  y := next(x);
  if( y ≠ NIL) then {
    if( k(x) ≤ k(y)) then sorted := F;
  }
  x := y;
}
@post: sorted = T ⇒ found = exists

```

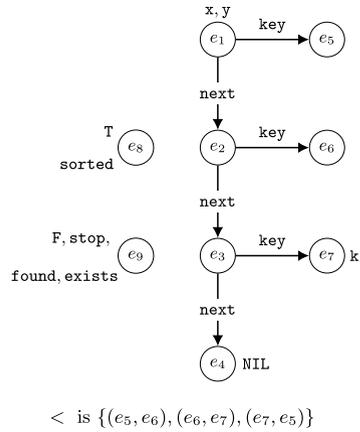


Fig. 1. Left: Uninterpreted program for finding a key k in a list starting at x with $<$ interpreted as a strict total order. The condition $a \leq b$ is shorthand for $a < b \vee a = b$. Right: A model in which $<$ is not interpreted as a strict total order. The elements in the universe of the model are denoted using circles. Some elements are labeled with variables denoting the initial values of these variables. The edges \rightarrow represent subterm relation. Not all functions are shown in the figure. The model does not satisfy the post-condition on the program on left.

beginning to end, and if it finds k , it sets a Boolean variable `exists` to T. Notice this algorithm does not exploit the sortedness property of the list. Algorithm 2 also walks through the list, but it stops as soon as it either finds k or reaches an element that is larger than k . If it finds the element it sets a Boolean variable `found` to T. If both algorithms are run on the same sorted list, then their answers (namely, `exists` and `found`) must be the same.

Fig. 1 (on the left) shows a program that weaves the above two algorithms together (treating Algorithm 1 as the specification for Algorithm 2). The variable `x` walks down the list using the `next` pointer. The variable `stop` is set to T when Algorithm 2 stops searching in the list. The precondition, namely that the input list is sorted, is captured by tracking another variable `sorted` whose value is T if consecutive elements are ordered as the list is traversed. The post condition demands that whenever the list is sorted, `found` and `exists` be equal when the list has been fully traversed. Note that the program’s correctness is specified using only quantifier-free assertions using the same vocabulary as the program.

The program works on a data domain that provides interpretations for the functions `key`, `next`, the initial values of the variables, and the relation $<$. When $<$ is interpreted to be a strict total order, the program is correct. However, if $<$ is not interpreted as a total order, then the program may be incorrectly deemed as buggy. To see this, consider the data model shown on the right in Fig. 1. The data domain has 9 elements in its universe, with the functions `next` and `key` interpreted as shown. Initially, `x, y` have value e_1 , `NIL` is e_4 , `k` is e_7 , `T` and `sorted` are e_8 , and `F, found, exists`, and `stop` are e_9 . The interpretation of $<$ is as follows — $e_5 < e_6$, $e_6 < e_7$, and $e_7 < e_5$. Clearly $<$ is not an order,

but the program's *sortedness* check “`sorted = T`” will pass. After the entire list is processed, `exists` will be set to `T` when $x = e_3$. On the other hand, `stop` will be set to `T` when $x = e_1$ because $k = e_7 < \text{key}(x)$. Therefore, at the end `found = F` \neq `exists`. The work presented in [26], where all functions and relations are uninterpreted, would therefore declare this program to be incorrect.

The goal of this paper is to explore several natural restrictions on data models and study the problem of verifying coherent programs for them. When $<$ is constrained to be a total order, the program in Fig. 1 is correct and coherent. Our results (see Section 5.5) show that verification of such programs when relations are constrained to be strict total orders is decidable, and hence we can build automatic decision procedures that will correctly verify such programs.

3 Preliminaries

We briefly recall the syntax and semantics of uninterpreted programs and the verification problem modulo axioms. Our presentation closely follows [26] and for lack of space, some details have been postponed to [27].

3.1 Program Syntax

We consider imperative programs with loops over a fixed finite set of variables V and use constant (\mathcal{C}), function (\mathcal{F}), and predicate (\mathcal{R}) symbols belonging to some first order signature $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$. Programs are then given by the syntax below ($f \in \mathcal{F}, R \in \mathcal{R}, x, y \in V, \mathbf{z}$ is a tuple of variables in V):

$$\begin{aligned} \langle \text{stmt} \rangle &::= | x := y \mid x := f(\mathbf{z}) \mid \mathbf{assume}(\langle \text{cond} \rangle) \mid \mathbf{skip} \mid \langle \text{stmt} \rangle ; \langle \text{stmt} \rangle \\ &\quad | \mathbf{while}(\langle \text{cond} \rangle) \langle \text{stmt} \rangle \mid \mathbf{if}(\langle \text{cond} \rangle) \mathbf{then} \langle \text{stmt} \rangle \mathbf{else} \langle \text{stmt} \rangle \\ \langle \text{cond} \rangle &::= x = y \mid R(\mathbf{z}) \mid \neg \langle \text{cond} \rangle \end{aligned}$$

3.2 Executions and Semantics of Uninterpreted Programs

Executions of programs over $\langle \text{stmt} \rangle$ are words over the following alphabet

$$\begin{aligned} \Pi = \{ & \text{“}x := y\text{”}, \text{“}x := f(\mathbf{z})\text{”}, \text{“}\mathbf{assume}(x = y)\text{”}, \text{“}\mathbf{assume}(x \neq y)\text{”}, \\ & \text{“}\mathbf{assume}(R(\mathbf{z}))\text{”}, \text{“}\mathbf{assume}(\neg R(\mathbf{z}))\text{”} \mid x, y \in V, \mathbf{z} \text{ is in tuples}(V) \} \end{aligned}$$

For a program $s \in \langle \text{stmt} \rangle$, the set of executions of s , denoted $\text{Exec}(s)$ is a regular language over the alphabet Π and is given as follows (similar to [26]).

$$\begin{aligned} \text{Exec}(\mathbf{skip}) &= \epsilon & \text{Exec}(x := y) &= \text{“}x := y\text{”} \\ \text{Exec}(x := f(\mathbf{z})) &= \text{“}x := f(\mathbf{z})\text{”} & \text{Exec}(\mathbf{assume}(c)) &= \text{“}\mathbf{assume}(c)\text{”} \\ \text{Exec}(\mathbf{if } c \mathbf{ then } s_1 \mathbf{ else } s_2) &= \text{“}\mathbf{assume}(c)\text{”} \cdot \text{Exec}(s_1) + \text{“}\mathbf{assume}(\neg c)\text{”} \cdot \text{Exec}(s_2) \\ \text{Exec}(s_1; s_2) &= \text{Exec}(s_1) \cdot \text{Exec}(s_2) \\ \text{Exec}(\mathbf{while } c \{ s \}) &= [\text{“}\mathbf{assume}(c)\text{”} \cdot \text{Exec}(s_1)]^* \cdot \text{“}\mathbf{assume}(\neg c)\text{”} \end{aligned}$$

The set of partial executions of s is the set of prefixes of words in $\text{Exec}(s)$ and is also regular.

A data model $\mathcal{M} = (U_{\mathcal{M}}, \llbracket \cdot \rrbracket_{\mathcal{M}})$ for signature Σ is a first order structure where $U_{\mathcal{M}}$ is a universe of elements and $\llbracket \cdot \rrbracket_{\mathcal{M}}$ maps every symbol in Σ to their interpretations. Given a data model \mathcal{M} over Σ , and an execution $\rho \in \Pi^*$, the semantics of ρ on \mathcal{M} is given by $\text{eval}_{\mathcal{M}} : \Pi^* \times V \rightarrow U_{\mathcal{M}}$ that gives the the valuation of variables in V at the end of an execution; the precise definition is standard and is deferred to [27].

3.3 Feasibility of Executions Modulo Axioms

An execution is said to be *feasible* in a data model, if every assumption made in the execution, holds on the model. More precisely, an execution ρ is feasible in \mathcal{M} if for every prefix $\sigma' = \sigma \cdot \text{"assume } c\text{"}$ of ρ , we have

- (a) $\text{eval}_{\mathcal{M}}(\sigma, x) = \text{eval}_{\mathcal{M}}(\sigma, y)$ if c is $\langle x = y \rangle$,
- (b) $\text{eval}_{\mathcal{M}}(\sigma, x) \neq \text{eval}_{\mathcal{M}}(\sigma, y)$ if c is $\langle x \neq y \rangle$,
- (c) $(\text{eval}_{\mathcal{M}}(\sigma, z_1), \dots, \text{eval}_{\mathcal{M}}(\sigma, z_r)) \in \llbracket R \rrbracket_{\mathcal{M}}$ if c is $\langle R(z_1, \dots, z_r) \rangle$, and
- (d) $(\text{eval}_{\mathcal{M}}(\sigma, z_1), \dots, \text{eval}_{\mathcal{M}}(\sigma, z_r)) \notin \llbracket R \rrbracket_{\mathcal{M}}$ if c is $\langle \neg R(z_1, \dots, z_r) \rangle$.

Let \mathcal{A} be a set of first order sentences, including possible ground atomic predicates². We say that a data model \mathcal{M} is an \mathcal{A} -model, denoted $\mathcal{M} \models \mathcal{A}$, if for every $\varphi \in \mathcal{A}$, we have $\mathcal{M} \models \varphi$. A formula φ is \mathcal{A} -valid, denoted $\mathcal{A} \models \varphi$, if ϕ holds in every model \mathcal{M} that satisfies \mathcal{A} . An execution ρ is said to be *feasible modulo* \mathcal{A} if there is an \mathcal{A} -model \mathcal{M} such that ρ is feasible in \mathcal{M} .

3.4 Program Verification Modulo Axioms

We consider programs annotated with post-conditions that are over the following syntax below. Here, x, y and \mathbf{z} belong to the set of program variables V and $R \in \mathcal{R}$ is a relation symbol in Σ .

$$\mathcal{L} : \quad \varphi ::= x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg \varphi$$

Definition 1 (Program Verification Modulo Axioms). *For a program s and a set of axioms \mathcal{A} , we say that s satisfies a postcondition φ over the syntax \mathcal{L} modulo \mathcal{A} if for every \mathcal{A} -model \mathcal{M} and for execution $\rho \in \text{Exec}(s)$ that is feasible in \mathcal{M} , \mathcal{M} satisfies $\varphi[\text{eval}_{\mathcal{M}}(\rho, V)/V]$ (i.e., where each variable $x \in V$ is replaced by $\text{eval}_{\mathcal{M}}(\rho, V)$).*

We remark that one can alternatively phrase the verification problem stated above in terms of feasibility. That is, a program s satisfies a postcondition φ modulo \mathcal{A} iff every execution ρ of s' is infeasible modulo \mathcal{A} (i.e., there is no \mathcal{A} -model \mathcal{M} such that ρ is feasible in \mathcal{M}), where $s' = s; \text{assume}(\neg \varphi)$.

² A ground atomic predicate is of the form $t_1 \sim t_2$, or $R(t_1, \dots, t_k)$ or $\neg R(t_1, \dots, t_k)$, where $\sim \in \{=, \neq\}$, R is a relation symbol, and t_i s are ground terms.

4 Coherence Modulo Axioms

In this section we extend the notion of coherence from [26], adapting it to our current setting where we restrict data models using axioms \mathcal{A} . We will first recall the notion of terms computed by an execution, which will be used to define the notion of coherence.

4.1 Terms Computed and Assumptions Accumulated by Executions

We will associate a syntactic term $\text{TEval}(\rho, x)$ with each variable $x \in V$ after a partial execution ρ . Intuitively, every variable $x \in V$ stores a constant term \hat{x} in the beginning of an execution. New terms are computed on function computations, i.e., $\text{TEval}(\rho \cdot “x := f(z_1, \dots, z_r)”) = f(\text{TEval}(\rho, z_1), \dots, \text{TEval}(\rho, z_r))$. The precise definition is simple and is deferred to [27]. The set of terms computed by an execution ρ is $\text{Terms}(\rho) = \{ \text{TEval}(\rho', x) \mid \rho' \text{ is a prefix of } \rho, x \in V \}$.

As an execution proceeds, it accumulates assumptions over the terms it computes, and we will use $\kappa(\rho)$ to denote the assumptions made by the execution ρ (see [27] for precise definition). For example, after an equality assume statement “**assume**($x = y$)”, we accumulate the atomic equality predicate $\psi = t_x = t_y$, where t_x and t_y are terms associated with x and y when the assume statement is encountered. Similarly, for the execution $\rho = \rho' \cdot “\text{assume}(\neg R(z_1, z_2, \dots, z_k))”$, we have $\kappa(\rho) = \kappa(\rho') \cup \{ \neg R(\text{TEval}(\rho', z_1), \dots, \text{TEval}(\rho', z_k)) \}$.

4.2 Coherence

Our definition of coherence modulo axioms is a smooth generalization of the definition of coherence in [26]. The notion of coherence consists of two properties — *memoizing* and *early equality assumes*. The memoizing property says, intuitively, when a term t is computed after executing some prefix σ of an execution, if t is equivalent to some other term modulo the assumptions made in the execution so far, then t must not have been *dropped* at the end of σ , i.e., a program variable must already hold this term. We replace the notion of equivalence of terms in this definition by equivalence modulo the axioms as well.

The notion of early assumes in [26] intuitively says that assumptions of equality (on terms t_1 and t_2) should be encountered early — earlier than *dropping* any superterm of t_1 or t_2 . This notion of early assumes allows for effectively computing *congruence closure* on the set of terms computed by the execution, which in turn, is necessary to accurately maintain which terms are equivalent. However, we observe that the notion in [26] is too restrictive and not entirely necessary. In our paper, we generalize this notion in several ways, to a more semantic one as follows. Whenever an execution encounters an assumption of equality between two term, we instead demand that only the equivalences that are *additionally* implied by this new assumption, can be inferred *locally* using the already known congruence between terms in the *window*, i.e., the set of terms pointed to by the program variables when the equality assumption is encountered. Next, we incorporate axioms into this definition, by requiring that the notion of equivalence is

also modulo the axioms, and further require that *all* assumptions (equality, disequality, relational) are required to be early (as against only restricting equality assumptions to be early like in [26]). We will elaborate on these differences using an example after presenting the formal definition next.

Given a set of first order sentences Γ and ground terms t_1 and t_2 , we say that $t_1 \cong_{\Gamma} t_2$ if $\Gamma \models t_1 = t_2$.

Definition 2 (Coherence modulo axioms). *Let \mathcal{A} be a set of axioms and let ρ be a complete or partial execution over variables V . Then, ρ is said to be coherent modulo \mathcal{A} if it satisfies the following two properties.*

Memoizing. *Let $\pi = \sigma \cdot "x := f(\mathbf{z})"$ be a prefix of ρ and let $t = \text{TEval}(\pi, x)$. If there is a term $t' \in \text{Terms}(\sigma)$ such that $t' \cong_{\mathcal{A} \cup \kappa(\sigma)} t$, then there must exist some variable $y \in V$ such that $\text{TEval}(\sigma, y) \cong_{\mathcal{A} \cup \kappa(\sigma)} t$.*

Early Assumes. *Let $\pi = \sigma \cdot "assume(c)"$ be a prefix of ρ , where c is any of $x=y$, $x \neq y$, $R(\mathbf{z})$, or $\neg R(\mathbf{z})$. Let $t \in \text{Terms}(\sigma)$ be a term computed in σ such that t has been dropped, i.e., for every $x \in V$, we have $\text{TEval}(\sigma, x) \not\cong_{\mathcal{A} \cup \kappa(\sigma)} t$. For any term $t' \in \text{Terms}(\sigma)$, if $t \cong_{\mathcal{A} \cup \kappa(\pi)} t'$, then $t \cong_{\mathcal{A} \cup \kappa(\sigma)} t'$.*

Remark. We remark that every execution that is coherent as per the definition in [26], is also coherent modulo $\mathcal{A} = \emptyset$ as in Definition 2. However, the converse is not true and we illustrate this difference below.

Example 1. Let us now illustrate the notion of coherence in the presence of axioms using the execution ρ below.

$$\rho = z_1 := f(x, y) \cdot z_2 := f(y, x) \cdot z_3 := g(z_1) \cdot z_4 := g(z_2) \cdot z_5 := z_5 \cdot z_6 := g(z_1)$$

Let ρ_i denote the prefix of ρ of length i . Here, $\text{TEval}(\rho_3, z_3) = g(f(\widehat{x}, \widehat{y}))$, $\text{TEval}(\rho_5, z_3) = \widehat{z}_5 \neq g(f(\widehat{x}, \widehat{y}))$ and $\text{TEval}(\rho_6, z_6) = g(f(\widehat{x}, \widehat{y}))$. When the set of axioms is $\mathcal{A} = \emptyset$, this execution is not coherent modulo \mathcal{A} as it violates the memoizing requirement at the last statement $z_6 := g(z_1)$ (no variable stores the term $g(f(\widehat{x}, \widehat{y}))$ after ρ_5).

Now, consider the axiom set denoting commutativity of f , i.e., $\mathcal{A}_{\text{comm}} = \{\forall u, v. f(u, v) = f(v, u)\}$. In this case, we observe that $f(\widehat{x}, \widehat{y}) \cong_{\mathcal{A}_{\text{comm}}} f(\widehat{y}, \widehat{x})$ and thus $g(f(\widehat{x}, \widehat{y})) \cong_{\mathcal{A}_{\text{comm}}} g(f(\widehat{y}, \widehat{x}))$. Also, $\text{TEval}(\rho_5, z_4) = g(f(\widehat{y}, \widehat{x})) \cong_{\mathcal{A}_{\text{comm}}} g(f(\widehat{x}, \widehat{y}))$. This ensures that ρ is indeed coherent modulo $\mathcal{A}_{\text{comm}}$.

Let $\text{CoherentExecs}(\Sigma, V, \mathcal{A})$ denote the set of executions over the signature Σ and variables V that are coherent modulo the set of axioms \mathcal{A} .

Definition 3. *A program s over signature Σ and variables V is said to be coherent modulo \mathcal{A} if $\text{Exec}(s) \subseteq \text{CoherentExecs}(\Sigma, V, \mathcal{A})$.*

In this paper, we explore several classes of axioms, studying when the verification problem for coherent programs modulo the axioms is decidable.

5 Axioms over Relations

In this section, we investigate the decidability of the verification problem for coherent programs modulo relational axioms, i.e., axioms which only involve relation symbols \mathcal{R} in the signature Σ .

5.1 Verification modulo EPR axioms

A first-order formula is said to be an EPR formula [37] if it is of the form

$$\exists x_1 \dots x_k \forall y_1, \dots y_m \varphi$$

where φ is quantifier-free and purely relational (uses no function symbols).

It is well known that satisfiability of EPR formulas is decidable, in fact by a reduction to Boolean satisfiability [24]. Consequently, the problem of checking whether a single execution is feasible under axioms written in EPR can be shown to be decidable, and has been exploited in bounded model-checking.

Consequently, we could reasonably ask whether verification of coherent programs under EPR axioms is decidable. Surprisingly, we show that they are not (proof details can be found in [27]).

Theorem 1. *Verification of uninterpreted coherent programs modulo EPR axioms is undecidable.* □

Given the above result, we turn to several classes of quantified axioms, which are all expressible in EPR (and hence have a decidable bounded model checking problem) and examine their decidability for coherent program verification.

5.2 Reflexivity, Irreflexivity, and Symmetry

We consider program verification under the following axioms (individually):

$$\begin{aligned} \varphi_{\text{refl}}^R &\triangleq \forall x \cdot R(x, x) && \text{(reflexivity)} \\ \varphi_{\text{irref}}^R &\triangleq \forall x \cdot \neg R(x, x) && \text{(irreflexivity)} \\ \varphi_{\text{symm}}^R &\triangleq \forall x, y \cdot R(x, y) \implies R(y, x) && \text{(symmetry)} \end{aligned} \tag{1}$$

We show that verification is decidable modulo these axioms using a technique that we call *program instrumentation*. Let us fix a relation R and an axiom φ_p^R , where $p \in \{\text{refl}, \text{irref}, \text{symm}\}$. The idea is to find a function (in fact, a string homomorphism) h_p^R such that for any program P , P is correct/coherent modulo $\{\varphi_p^R\}$ iff $h_p^R(\text{Exec}(P))$ is correct/coherent modulo the empty axiom set. Decidability then follows by exploiting the results of [26]. The function h_p^R will capture the properties of the axiom it is trying to eliminate, and so it will be different for different axioms. We first outline these function h_p^R , then state their property and prove the decidability result.



Fig. 2. Implied negative relational assumes for a transitive relation R . The dashed edges ($--\rightarrow$) represent the inferred relationship implied from the relations marked by bold edges (\rightarrow).

For reflexivity, we transform an execution ρ of P to ρ' where ρ' is essentially ρ , except that whenever we see the computation of a term, using an assignment of the form “ $x := f(\mathbf{z})$ ”, we immediately insert an assume statement that states that $R(x, x)$ holds. More precisely, the homomorphism is defined as,

$$h_{\text{refl}}^R(a) = \begin{cases} a \cdot \text{“assume}(R(x, x))\text{”} & \text{if } a = \text{“}x := f(\mathbf{z})\text{”} \\ a & \text{otherwise} \end{cases}$$

The homomorphisms used for irreflexivity and symmetry follow similar lines and are outlined in [27].

Theorem 2. *For any relation symbol R and $p \in \{\text{refl}, \text{irref}, \text{symm}\}$, the problems of coherent verification modulo $\{\varphi_p^R\}$ and checking coherence modulo $\{\varphi_p^R\}$ are PSPACE-complete.*

5.3 Transitivity

We now consider the transitivity axiom for a relation R which says

$$\varphi_{\text{trans}}^R = \forall x, y, z \cdot R(x, y) \wedge R(y, z) \implies R(x, z) \quad (\text{transitivity}) \quad (2)$$

The proof for decidability modulo this axiom is different and more complex than the proofs for reflexivity, irreflexivity, and symmetry. Intuitively, the program instrumentation approach does not seem to work for transitivity. This is because transitivity effects can be global. For example, we may have that the execution asserts the sequence of relational assumes $R(t_1, t_2), R(t_2, t_3), \dots, R(t_{n-1}, t_n)$ (here, t_1, \dots, t_n are terms computed by the execution), where some of the intermediate terms may have been dropped by the program (i.e., the variables holding these terms were reassigned). Consequently, relating t_1 and (the possibly newly constructed term) t_n requires a principally new machinery. We modify the automaton construction from [26] so that it maintains the transitive closure of the assumptions the program makes. Our main observation is the following:

Theorem 3. *Let Σ be a first order signature and V a finite set of program variables. Let $\mathcal{A} = \{\varphi_{\text{trans}}^R \mid R \in \mathcal{R}_{\text{trans}}\}$ for some set of relation symbol $\mathcal{R}_{\text{trans}}$ in Σ . The following observation hold.*

1. There is a finite automaton \mathcal{F}_{trans} (effectively constructable) of size $O(2^{poly(|V|)})$ such that for any coherent execution ρ that is coherent modulo \mathcal{A} , \mathcal{F}_{trans} accepts ρ iff ρ is feasible.
2. There is a finite automaton \mathcal{C}_{trans} (effectively constructible) of size $O(2^{poly(|V|)})$ such that $L(\mathcal{C}_{trans}) = \text{CoherentExecs}(\Sigma, V, \mathcal{A})$.

Proof Sketch. These are in some sense a generalization of the automata constructions used to establish decidability in [26]. The automata \mathcal{F}_{trans} and \mathcal{C}_{trans} rely on tracking equivalence between values stored in variables, and functional and relational correspondences between these values. However, now since some relations maybe transitive, additional relational correspondences (or their absence) maybe implied for $R \in \mathcal{R}_{trans}$. The basic idea is to maintain for transitive relations R (a) the transitive closure of the positive relation assumes **assume**($R(\cdot, \cdot)$), and (b) the negative relational assumes implied by the relational assumes seen in an execution. More precisely, if the execution sees **assume**($R(x, y)$) and **assume**($R(y, z)$), then we also add the constraint $R(x, z)$ in the automaton's state. Further, if the execution observes **assume**($R(x, y)$) and **assume**($\neg R(x, z)$), then one can infer the constraint $\neg R(y, z)$, and in this case, we accumulate this additional constraint in the state of the automaton. Similarly, if the execution observes **assume**($R(y, z)$) and **assume**($\neg R(x, z)$), then one can infer the constraint $\neg R(x, y)$, which is added in the automaton's state. Both these scenarios are illustrated in Fig. 2. A detailed proof is in [27]. \square

As a consequence we have the following result.

Theorem 4. For $\mathcal{A} = \{\varphi_{trans}^R \mid R \in \mathcal{R}_{trans}\}$, the problems of coherent verification modulo \mathcal{A} and checking coherence modulo \mathcal{A} are PSPACE-complete.

5.4 Strict Partial Orders

We now turn our attention to axioms that dictate that certain relations be partial or total orders. The anti-symmetry axiom that holds for non-strict orders introduces subtle complications. Recall that R is anti-symmetric if $\forall x, y. R(x, y) \wedge R(y, x) \Rightarrow x = y$; this axiom can imply equality between terms if R holds between a pair of terms. Concretely, if R is anti-symmetric, and the program makes assumptions in an execution that $R(t_1, t_2)$ and $R(t_2, t_1)$ hold, then any model in which such an execution is feasible must also ensure that $t_1 = t_2$. This implicit equality assumption interferes with the notions of coherence and the automata constructions (proofs of the results in [26] and Theorem 4) that compute a congruence closure on terms in a streaming fashion.

Hence, we only consider *strict* partial orders in this section. Recall that a relation R is a strict partial order if it satisfies the irreflexivity axiom and the transitivity axiom, together denoted \mathcal{A}_{SPO}^R . We can prove decidability for problems modulo \mathcal{A}_{SPO}^R by using our algorithm for irreflexivity and transitivity.

Theorem 5. The following problems are PSPACE-complete.

1. Given a program P that is coherent modulo \mathcal{A}_{SPO}^R , determine if P is correct.
2. Given a program P , determine if P is coherent modulo \mathcal{A}_{SPO}^R

5.5 Strict Total Orders

A relation R is a strict total order if it is a strict partial order and satisfies:

$$\forall x, y \cdot x \neq y \implies R(x, y) \vee R(y, x) \quad (\text{totality}) \quad (3)$$

Strict total orders are again tricky to handle as the axiom for totality can result in implicit equality between terms. For example, if $\neg R(x, y)$ and $\neg R(y, x)$ then it must be the case that $x = y$. However, if we restrict ourselves to executions that only have assumes of the form **assume**($R(x, y)$) and do not have any assumes on $\neg R$, i.e., of the form **assume**($\neg R(x, y)$) then there are no implicit equalities that are entailed.

Unfortunately, in general, program executions can contain negative assumes on R (i.e., assumes of the form **assume**($\neg R(x, y)$)). In order to ensure that executions contain only *positive* assumptions on R , we must be careful when identifying executions of programs with conditionals — branches where the assumption $\neg R(x, y)$ holds must be translated to a branch that assumes $R(y, x)$ and a branch that assumes $x = y$. We present a detailed translation in [27].

After such a translation, executions can now have additional equality assumes even if they did not appear in the program. When we refer to coherent programs, we mean that they are coherent according to the above modified notion of executions. This means for such programs to be coherent, all executions must ensure that the additional equality assumes are *early*. And when we talk about coherent verification of programs with total orders, we mean verification for programs that are coherent after this transformation.

We observe that in the absence of any assumes of the form $\neg R(x, y)$ the verification problem modulo strict total orders reduces that modulo strict partial orders, giving us the following (A_{STO}^R denote the axioms of irreflexivity, transitivity and totality for the relation R).

Theorem 6. *The problems of coherent verification, and checking coherence modulo A_{STO}^R are PSPACE-complete.*

6 Axioms Over Functions

We now discuss computational problems modulo axioms that involve function symbols. The treatment of axioms involving functions in the verification of coherent programs is inherently hard. This is because, like in the case of (nonstrict) partial orders and strict total orders, the axioms along with the **assume**-steps in the execution, can imply equalities between terms beyond those entailed by just the **assume** steps in the execution. For example, consider the axiom $\forall x, y \cdot f(x, y) = f(y, x)$ constraining f to be a commutative function. Then terms like $f(f(x, y), z)$ are equal to terms like $f(z, f(x, y))$, and hence when building models we must make sure that functions/relations on such terms are defined in the same way. Terms made equivalent by the functional axioms can be syntactically very different, and keeping track of the equivalence on unbounded

executions is hard using finite memory. We consider many natural classes of axioms, and proving both positive and negative results that help delineate the decidability/undecidability boundary.

6.1 Associativity

We now consider the associativity axiom for a function f .

$$\varphi_{\text{assoc}}^f \triangleq \forall x, y, z \cdot f(x, f(y, z)) = f(f(x, y), z) \quad (\text{associativity}) \quad (4)$$

We show, surprisingly to us, that coherent verification is undecidable modulo $\{\varphi_{\text{assoc}}^f\}$, i.e., even when we have only one axiom that requires only one function to be associative. In fact, the situation is a lot worse — checking the feasibility of even a *single* (even coherent) execution is undecidable, in the presence of a single associative function. The proof of the following result uses a reduction from the word problem for finitely generated semigroups [36].

Theorem 7. *Given a trace ρ that is coherent modulo $\{\varphi_{\text{assoc}}^f\}$, it is undecidable to determine if ρ is feasible. Therefore, the problem checking if a program P that is coherent modulo $\{\varphi_{\text{assoc}}^f\}$ is undecidable.*

6.2 Commutativity

We now consider the commutativity axiom, which is the following

$$\varphi_{\text{comm}}^f \triangleq \forall x, y \cdot f(x, y) = f(y, x) \quad (\text{commutativity}) \quad (5)$$

We augment executions with an auxiliary variable $v^* \notin V$ and transform executions using the following homomorphism that uses the auxiliary variable v^*

$$h_{\text{comm}}^f(a) = \begin{cases} a \cdot \text{“}v^* := f(y, x)\text{”} \cdot \text{“}\mathbf{assume}(z = v^*)\text{”} & \text{if } a = \text{“}z := f(x, y)\text{”} \\ a & \text{otherwise} \end{cases}$$

We show that the above transformation preserves feasibility and coherence, giving us the following result.

Theorem 8. *Verification of coherent programs and checking coherence modulo commutativity axioms is decidable and is PSPACE-complete.*

6.3 Idempotence

Next we consider the idempotence axiom for a unary function f :

$$\varphi_{\text{idem}}^f \triangleq \forall x \cdot f(x) = f(f(x)) \quad (\text{idempotence}) \quad (6)$$

Again, we show that there is a simple homomorphism h_{idem}^f that preserves coherence and feasibility (see [27]) and reduces verification to one without axioms.

Theorem 9. *Verification of coherent programs and checking coherence modulo idempotence axioms is PSPACE-complete.*

7 Combining Axioms

We have thus far proved decidability results when a relation or functions satisfies certain properties like reflexivity/irreflexivity/symmetry/transitivity or commutativity/idempotence. We now show that all of these results can be combined. That is, we can consider a signature where relations and functions are assumed to satisfy some subset of these properties, and with some being uninterpreted, and the verification problem will remain decidable for coherent programs.

Theorem 10. *Let \mathcal{A} be a set of axioms where each relation symbol R is either a total order or satisfies some (possibly empty) subset of properties out of reflexivity, irreflexivity, symmetry, transitivity, and each function symbol f satisfies some (possibly empty) subset out of commutativity and idempotence. The verification problem for coherent programs modulo \mathcal{A} is PSPACE-complete.*

The proof of the above result proceeds by *eliminating* axioms one at a time. We first eliminate the relational axioms (reflexivity, irreflexivity, symmetry) in \mathcal{A} using program instrumentation. We then eliminate the functional axioms in \mathcal{A} , again using program instrumentation. Our proof relies on this order of elimination of axioms. At this point, the only axioms remaining are those corresponding to transitivity of a subset of relational symbols, which is handled using the automata construction discussed in the proof of Theorem 3.

8 Related Work

The theory of equality with uninterpreted functions (EUF) is a widely used theory in many verification applications as it has decidable quantifier free fragment. EUF has been central to advances in verification of microprocessor control [6,4] and hardware verification [1,19] and property directed model checking [18]. EUF has been used as a popular abstraction in software verification [2,3]. Uninterpreted functions have also been studied for equivalence checking and translation validation [35]. Bueno et al [5] demonstrated the effectiveness of uninterpreted programs for verifying SVCOMP benchmarks against control flow properties.

Mathur et al [26] introduced the class of coherent uninterpreted programs and showed that verification of coherent programs, with or without recursive function calls, is a decidable problem. This is one of the few subclasses of program verification over infinite domains that is known to be decidable. Previous works [13,14,31] have established decidability of verification of classes of uninterpreted programs with heavy syntactic restrictions such as disallowing conditionals inside loops or nested loops, etc. As noted in [26], the notion of coherence is close to the notion of a bounded pathwidth decomposition [38]. A term that is created in a coherent execution stays within some program variable (modulo congruence) until the first time all variables containing that term are over-written, and after this point, the execution never computes it again, and thus, the set of windows that contain a term form a contiguous segment of the program execution. Path decomposition and the related notion of tree decomposition have been exploited many times in the literature to give decidability in verification [25,7,8].

The work in [28] extends the work of [26] to *updatable maps* and identifies extensions of coherence that make verification decidable. It utilizes this to provide implementation of verification algorithms for memory safety for a class of heap manipulating programs, including traversal algorithms on data structures such as singly linked list, sorted lists, binary search trees etc. Combining the results of this paper with these results is an interesting future direction.

The class of EPR formulas that consist of universally quantified formulas over relational signatures is a well-known decidable class of first-order logic [37]. EPR-based reasoning has been proved powerful for verification of large-scale systems [33,29,39] and the Ivy [34,30] system is one of the most notable framework that exploits EPR based reasoning for verifying program snippets without recursion. EPR encoding of order axioms such as reflexivity, symmetry, transitivity and total orders has been used in proving programs working over heaps [20].

The work in Kleene Algebra with Tests (KAT) [22] considers problems involving unbounded recursion and choice with abstractions of data, similar to our work. However, while we treat congruence axioms for equality faithfully in our work, it is unclear to us how to express these in KAT or its extensions [21,23,9]. Furthermore, the restrictions of coherence studied in [26] and the work here that are based on bounded path-width notions seem very different from studies of decidable problems in KAT. A study of whether our results can be adapted to yield decidable fragments for KAT is an interesting future direction.

A notable verification technique with an automata-theoretic foundation and that has been very effective in practice is that of trace abstraction due to Heizmann et al [15,16,17,10,11,12]. In this technique, one constructs *iteratively* regular sets that (incompletely) capture the set of all infeasible executions, eventually striving to cover all failing executions of a program, but handling complex theories such as arithmetic. In contrast, our work builds complete automata in one stroke that accept all infeasible traces over a vocabulary, but handles only simple theories with restricted sets of axioms, but yielding decidability. Combining these lines of work for efficient software verification is an interesting future direction.

9 Conclusions

By incorporating axioms on functions and relations, decidability results in this paper, enable a more faithfully automatic verification of programs. It is worth noting that the upper bound for all our decidability results is PSPACE, which is the same as that for Boolean programs. Thus, though we consider programs over infinite domains with additional structure, our verification results have the same complexity as that for programs over Boolean domains.

One future direction is to adapt this technique for practical program verification. In this context, adapting our technique within the automata-theoretic technique of [15,17,16,12,10] seems most promising. Second, there are several program verification techniques that use EPR, and in several of these, EPR is used mainly to establish a linear order on the universe [20]. Automatically verifying such programs using our technique is worth exploring.

References

1. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Reveal: A formal verification tool for verilog designs. In: Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 343–352. LPAR '08, Springer-Verlag, Berlin, Heidelberg (2008)
2. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In: Proceedings of the 19th Int. Conf. on Computer Aided Verification (CAV'07), Berlin, Germany. Lecture Notes in Computer Science, Springer (July 2007)
3. Babic, D., Hu, A.J.: Calysto: Scalable and precise extended static checking. In: Proceedings of the 30th International Conference on Software Engineering. p. 211–220. ICSE '08, Association for Computing Machinery, New York, NY, USA (2008). <https://doi.org/10.1145/1368088.1368118>
4. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Proceedings of the 14th International Conference on Computer Aided Verification. pp. 78–92. CAV '02, Springer-Verlag, London, UK, UK (2002)
5. Bueno, D., Sakallah, K.A.: euforia: Complete software model checking with uninterpreted functions. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 363–385. Springer International Publishing, Cham (2019)
6. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Proceedings of the 6th International Conference on Computer Aided Verification. pp. 68–80. CAV '94, Springer-Verlag, London, UK, UK (1994)
7. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 733–747. POPL '16, ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2837614.2837624>
8. Chatterjee, K., Ibsen-Jensen, R., Pavlogiannis, A., Goyal, P.: Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 97–109. POPL '15, ACM, New York, NY, USA (2015). <https://doi.org/10.1145/2676726.2676979>
9. Doumane, A., Kuperberg, D., Pous, D., Pradic, P.: Kleene algebra with hypotheses. In: Bojańczyk, M., Simpson, A. (eds.) Foundations of Software Science and Computation Structures. pp. 207–223. Springer International Publishing, Cham (2019)
10. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 129–142. POPL '13, ACM, New York, NY, USA (2013). <https://doi.org/10.1145/2429069.2429086>
11. Farzan, A., Kincaid, Z., Podelski, A.: Proofs that count. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 151–164. POPL '14, ACM, New York, NY, USA (2014). <https://doi.org/10.1145/2535838.2535885>
12. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 407–420. POPL '15, ACM, New York, NY, USA (2015). <https://doi.org/10.1145/2676726.2677012>

13. Godoy, G., Tiwari, A.: Invariant checking for programs with procedure calls. In: Proceedings of the 16th International Symposium on Static Analysis. pp. 326–342. SAS '09, Springer-Verlag, Berlin, Heidelberg (2009)
14. Gulwani, S., Tiwari, A.: Assertion checking unified. In: Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 363–377. VMCAI'07, Springer-Verlag, Berlin, Heidelberg (2007)
15. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proceedings of the 16th International Symposium on Static Analysis. pp. 69–85. SAS '09, Springer-Verlag, Berlin, Heidelberg (2009)
16. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 471–482. POPL '10, ACM, New York, NY, USA (2010). <https://doi.org/10.1145/1706299.1706353>
17. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 36–52. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
18. Ho, Y.S., Mishchenko, A., Brayton, R.: Property directed reachability with word-level abstraction. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 132–139. FMCAD '17, FMCAD Inc, Austin, TX (2017). <https://doi.org/10.23919/FMCAD.2017.8102251>
19. Hojati, R., Isles, A., Kirkpatrick, D., Brayton, R.K.: Verification using uninterpreted functions and finite instantiations. In: Srivas, M., Camilleri, A. (eds.) Formal Methods in Computer-Aided Design. pp. 218–232. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
20. Itzhaky, S., Banerjee, A., Immerman, N., Lahav, O., Nanevski, A., Sagiv, M.: Modular reasoning about heap paths via effectively propositional formulas. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 385–396. POPL '14, ACM, New York, NY, USA (2014). <https://doi.org/10.1145/2535838.2535854>
21. Kozen, D.: Kleene algebra with tests and commutativity conditions. In: Margaria, T., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 14–33. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
22. Kozen, D.: Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.* **19**(3), 427–443 (May 1997). <https://doi.org/10.1145/256167.256195>
23. Kozen, D., Mamouras, K.: Kleene algebra with equations. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) Automata, Languages, and Programming. pp. 280–292. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
24. Lewis, H.: Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences* **21**(3), 317–353 (1980)
25. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 283–294. POPL '11, ACM, New York, NY, USA (2011). <https://doi.org/10.1145/1926385.1926419>
26. Mathur, U., Madhusudan, P., Viswanathan, M.: Decidable verification of uninterpreted programs. *Proc. ACM Program. Lang.* **3**(POPL), 46:1–46:29 (Jan 2019). <https://doi.org/10.1145/3290359>
27. Mathur, U., Madhusudan, P., Viswanathan, M.: What's decidable about program verification modulo axioms? *CoRR* **abs/1910.10889** (2019), <http://arxiv.org/abs/1910.10889>

28. Mathur, U., Murali, A., Krogmeier, P., Madhusudan, P., Viswanathan, M.: Deciding memory safety for single-pass heap-manipulating programs. *Proc. ACM Program. Lang.* **4**(POPL) (Dec 2019). <https://doi.org/10.1145/3371103>
29. McMillan, K.: Modular specification and verification of a cache-coherent interface. In: *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*. pp. 109–116. FMCAD '16, FMCAD Inc, Austin, TX (2016)
30. McMillan, K.L., Padon, O.: Deductive verification in decidable fragments with ivy. In: Podelski, A. (ed.) *Static Analysis*. pp. 43–55. Springer International Publishing, Cham (2018)
31. Müller-Olm, M., Rütting, O., Seidl, H.: Checking herbrand equalities and beyond. In: *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*. pp. 79–96. VMCAI'05, Springer-Verlag, Berlin, Heidelberg (2005)
32. Padon, O., Immerman, N., Shoham, S., Karbyshev, A., Sagiv, M.: Decidability of inferring inductive invariants. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 217–231. POPL '16, ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2837614.2837640>
33. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made epr: Decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.* **1**(OOPSLA), 108:1–108:31 (Oct 2017). <https://doi.org/10.1145/3140568>
34. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 614–630. PLDI '16, ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2908080.2908118>
35. Pnueli, A., Strichman, O.: Reduced functional consistency of uninterpreted functions. *Electron. Notes Theor. Comput. Sci.* **144**(2), 53–65 (Jan 2006). <https://doi.org/10.1016/j.entcs.2005.12.006>
36. Post, E.L.: Recursive unsolvability of a problem of thue. *J. Symbolic Logic* **12**(1), 1–11 (03 1947)
37. Ramsey, F.P.: *On a Problem of Formal Logic*, pp. 1–24. Birkhäuser Boston, Boston, MA (1987)
38. Robertson, N., Seymour, P.D.: Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B* **35**(1), 39–61 (1983)
39. Taube, M., Losa, G., McMillan, K.L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J.R., Woos, D.: Modularity for decidability of deductive verification with applications to distributed systems. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 662–677. PLDI 2018, ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3192366.3192414>

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

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





Formalized Proofs of the Infinity and Normal Form Predicates in the First-Order Theory of Rewriting^{*}

Alexander Lochmann[✉] and Aart Middeldorp[✉]

Department of Computer Science, University of Innsbruck, Austria
{alexander.lochmann,aart.middeldorp}@uibk.ac.at

Abstract. We present a formalized proof of the regularity of the infinity predicate on ground terms. This predicate plays an important role in the first-order theory of rewriting because it allows to express the termination property. The paper also contains a formalized proof of a direct tree automaton construction of the normal form predicate, due to Comon.

Keywords: Formalization · First-order theory of rewriting · Tree automata

1 Introduction

Term rewriting [1, 18] is an abstract model of computation which underlies much of declarative programming and automated theorem proving. The foundation of rewriting is equational logic. Equations are used from left to right to direct the search for proofs. Fundamental properties like *confluence* (which ensures that different computation paths produce the same result) and *termination* (all computation paths produce a result) are *undecidable* in general. For terminating systems, one is interested in estimating the resources needed to evaluate expressions (space and time *complexity*). Much progress has been made in establishing sufficient and automatable criteria for confluence, termination, complexity, and other properties of rewrite systems. These criteria have been implemented in highly optimized automatic tools that compete on a yearly basis [12, 13]. These competitions, together with the recent advances in SAT [4] and SMT [2] solving, have on the one hand led to specialized techniques that are especially suitable for automation. On the other hand, software bugs observed in the tools gave rise to the more recent activity of *certification* of the output of termination, complexity, and confluence tools. This is done by formalizing the underlying methods in an interactive proof assistant like Coq [3] or Isabelle [15], and using the code generation facilities of these proof assistants to obtain trustworthy programs that can certify the output of the tools.

In this paper we are concerned with the formalization of methods that are used in FORT [16, 17], a tool that implements the first-order theory of rewriting

^{*} This research is supported by FWF (Austrian Science Fund) project P30301.

for the decidable class of left-linear, right-ground rewrite systems. FORT can be used to decide properties of a given rewrite system and to synthesize rewrite systems that satisfy arbitrary properties expressible in the first-order theory of rewriting. The decision procedure is based on tree automata techniques and goes back to a paper by Dauchet and Tison [7]. In a recent paper [10] the authors formalized results concerning ground tree transducers and RR_n automata for a fragment of the first-order theory that allows to express confluence, resulting in a formalized confluence prover for left-linear, right-ground rewrite systems. In this paper we cover the infinity predicate that is crucial for expressing the termination property in the first-order theory of rewriting and an efficient automaton construction of the normal form predicate that is employed in FORT. The former goes back to a technical report by Dauchet and Tison [8] and the latter is based on a paper by Comon [5]. The normal form predicate has other applications as well (e.g. [9, 14]). A proof of the construction of [8] is given in [16], but this proof contains a serious mistake that we report at the end of Section 3.

Our formalizations are based on *IsaFoR* [19],¹ an Isabelle/HOL library containing numerous abstract results and concrete techniques from the rewriting literature. Our own development can be found at

<http://cl-informatik.uibk.ac.at/software/fortissimo/tacas2020/>

Most definitions, theorems, and lemmata in this paper directly correspond to the formalization. These are indicated by the  symbol, which links to a HTML presentation in the PDF version of the paper.

In the next section we recall basic definitions, notation, and results concerning term rewriting and tree automata that we need in the sequel. In Section 3 we present our first main result, a formalized correctness proof of the regularity of the infinity predicate for regular relations. The tree automaton constructed in the correctness proof is not directly executable due to the definition of Q_∞ which plays an important role in the construction of the tree automaton. In Section 4 we present our second main result, an equivalent definition of Q_∞ that is constructive. Our third result, a formalized correctness proof of an efficient tree automata construction of the normal form predicate for left-linear rewrite systems, is the topic of Section 5. We conclude in Section 6 with some statistics of our formalizations as well as a list of tasks that remain to be done for a certified version of FORT.

When we write “formalized” we always mean “formalized in Isabelle/HOL.”

2 Preliminaries

Familiarity with term rewriting [1] and tree automata [6] is useful, but we briefly recall important definitions and notation that we use in the remainder.

We assume a given signature \mathcal{F} and a set of variables \mathcal{V} . Function symbols in \mathcal{F} are equipped with a fixed arity. Function symbols of arity zero are called

¹ <http://cl-informatik.uibk.ac.at/isafor/>

constants. The set of terms built from \mathcal{F} and \mathcal{V} is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and inductively defined: A term is either a variable $x \in \mathcal{V}$ or $f(t_1, \dots, t_n)$ for a function symbol f of arity n and terms $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of variables occurring in a term t is denoted by $\mathcal{Var}(t)$. A term t with $\mathcal{Var}(t) = \emptyset$ is called ground. We write $\mathcal{T}(\mathcal{F})$ for the set of ground terms. Positions are strings of positive integers which are used to address subterms. The empty string is called root position and denoted by ϵ . The set of positions in a term t is denoted by $\mathcal{Pos}(t)$ and the subterm of t at position $p \in \mathcal{Pos}(t)$ by $t|_p$. We write $s \triangleleft t$ if s is a proper subterm of t , i.e., $s = t|_p$ with $p \neq \epsilon$. We write $t[u]_p$ for the result of replacing the subterm of t at position p with the term u . The root symbol of a term t is denoted by $\text{root}(t)$ and $t(p)$ denotes $\text{root}(t|_p)$. We write $p < q$ if p is a proper prefix of q . A context C is a term with a hole \square . Here $\square \notin \mathcal{F}$ is a special constant. We write $C[t]$ for the result of replacing the hole in C by t . A substitution σ is a mapping from variables to terms. We write $t\sigma$ for the result of applying σ to the term t .

A term rewrite system (TRS for short) \mathcal{R} consists of rewrite rules $\ell \rightarrow r$ between terms ℓ and r over the same signature \mathcal{F} such that $\mathcal{Var}(r) \subseteq \mathcal{Var}(\ell)$. The rewrite relation $\rightarrow_{\mathcal{R}}$ is defined on terms as follows: $s \rightarrow_{\mathcal{R}} t$ if there exist a position $p \in \mathcal{Pos}(s)$, a rewrite rule $\ell \rightarrow r \in \mathcal{R}$, and a substitution σ such that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$. The reflexive transitive closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\rightarrow_{\mathcal{R}}^*$. A redex is a substitution instance of a left-hand side of a rewrite rule. Terms that contain a redex as subterm are called reducible. A normal form is a term without redexes. We write $\text{NF}(\mathcal{R})$ for the set of ground normal forms of \mathcal{R} . In this paper we consider finite TRSs over finite signatures. The TRSs handled by FORT are left-linear (no duplicate variables in left-hand sides of rewrite rules) and right-ground (no variables in right-hand sides of rewrite rules).

We now recall some basic notions related to tree automata. A *tree automaton* is a quadruple $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ consisting of a finite signature \mathcal{F} , a finite set Q of states, disjoint from \mathcal{F} , a subset $Q_f \subseteq Q$ of final states, and a set of transition rules Δ . Every transition rule has one of the following two shapes:

- $f(p_1, \dots, p_n) \rightarrow q$ with $f \in \mathcal{F}$ and $p_1, \dots, p_n, q \in Q$, or
- $p \rightarrow q$ with $p, q \in Q$.

Transition rules of the second shape are called epsilon transitions. We write Δ_ϵ for the set of epsilon transitions. Furthermore, $\Delta_{-\epsilon} = \Delta \setminus \Delta_\epsilon$. Transition rules can be viewed as rewrite rules between ground terms in $\mathcal{T}(\mathcal{F} \cup Q)$. The induced rewrite relation is denoted by \rightarrow_{Δ} or $\rightarrow_{\mathcal{A}}$. A ground term $t \in \mathcal{T}(\mathcal{F})$ is *accepted* by \mathcal{A} if $t \rightarrow_{\Delta}^* q$ for some $q \in Q_f$. The set of all accepted terms is denoted by $L(\mathcal{A})$ and a set L of ground terms is *regular* if $L = L(\mathcal{A})$ for some tree automaton \mathcal{A} .

Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton. A state $q \in Q$ is *reachable* if $t \rightarrow_{\Delta}^* q$ for some term $t \in \mathcal{T}(\mathcal{F})$. We say that q is *productive* if $C[q] \rightarrow_{\Delta}^* q_f$ for some ground context C and final state $q_f \in Q_f$. The automaton \mathcal{A} is *trim* if all states are both reachable and productive. Any tree automaton can be transformed into an equivalent trim automaton. This result has been formalized in IsaFoR by Felgenhauer and Thiemann [11].

Below we present a formalized proof of a version of the *pumping lemma* that we need later.

Lemma 1. *Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton and $t \rightarrow_{\Delta}^* q$ with $t \in \mathcal{T}(\mathcal{F})$ and $q \in Q$. If $\text{height}(t) > |Q|$ then there exist contexts C_1 and $C_2 \neq \square$, a term u , and a state p such that $t = C_1[C_2[u]]$, $u \rightarrow_{\Delta}^* p$, $C_2[p] \rightarrow_{\Delta}^* p$, and $C_1[p] \rightarrow_{\Delta}^* q$.*

Proof. From the assumptions $t \rightarrow_{\Delta}^* q$ and $\text{height}(t) > |Q|$ we obtain a sequence $(t_1, \dots, t_{n+1}, q_1, \dots, q_{n+1}, D_1, \dots, D_n)$ consisting of ground terms, states, and non-empty contexts with $n > |Q|$ such that

- $t_i \rightarrow_{\Delta}^* q_i$ for all $i \leq n + 1$,
- $D_i[t_i] = t_{i+1}$ and $D_i[q_i] \rightarrow_{\Delta}^* q_{i+1}$ for all $i \leq n$, and
- $q_{n+1} = q$ and $t_{n+1} = t$

by a straightforward induction proof on t . Because $n > |Q|$ there exist indices $1 \leq i < j \leq n$ such that $q_i = q_j$. We construct the contexts $C_1 = D_n[\dots[D_j]\dots]$ and $C_2 = D_{j-1}[\dots[D_i]\dots]$. Note that $C_2 \neq \square$ as $i < j$. We obtain $C_2[q_i] \rightarrow_{\Delta}^* q_j$ and $C_1[q_j] \rightarrow_{\Delta}^* q_{n+1}$ by induction on the difference $j - i$. By letting $p = q_i = q_j$ and $u = t_i$ we obtain the desired result. \square

We conclude this preliminary section with a brief account of RR_2 relations, which are binary relations on ground terms over a signature \mathcal{F} whose *encoding* as sets of ground terms over the extended signature $\mathcal{F}^{(2)} = (\mathcal{F} \cup \{\perp\})^2$ with a fresh constant $\perp \notin \mathcal{F}$ is regular. The arity of a symbol $fg \in \mathcal{F}^{(2)}$ is the maximum of the arities of f and g . The encoding of two terms $t, u \in \mathcal{T}(\mathcal{F})$ is the unique term $\langle t, u \rangle \in \mathcal{T}(\mathcal{F}^{(2)})$ such that $\text{Pos}(\langle t, u \rangle) = \text{Pos}(t) \cup \text{Pos}(u)$ and $\langle t, u \rangle(p) = fg$ where

$$f = \begin{cases} t(p) & \text{if } p \in \text{Pos}(t) \\ \perp & \text{otherwise} \end{cases} \quad g = \begin{cases} u(p) & \text{if } p \in \text{Pos}(u) \\ \perp & \text{otherwise} \end{cases}$$

for all positions $p \in \text{Pos}(t) \cup \text{Pos}(u)$. We illustrate this on a concrete example. For the ground terms $t = f(\mathbf{g}(\mathbf{a}), f(\mathbf{b}, \mathbf{a}))$ and $u = f(\mathbf{a}, \mathbf{g}(\mathbf{g}(\mathbf{b})))$ we obtain $\langle t, u \rangle = \text{ff}(\mathbf{g}(\mathbf{a}(\mathbf{a}\perp), \mathbf{f}(\mathbf{b}(\mathbf{g}(\perp\mathbf{b}), \mathbf{a}\perp)))$. A tree automaton operating on terms in $\mathcal{T}(\mathcal{F}^{(2)})$ is called an RR_2 automaton. The two *projection* operations effectively transform RR_2 relations on $\mathcal{T}(\mathcal{F})$ to regular subsets of $\mathcal{T}(\mathcal{F})$.

3 Infinity Predicate

The following formula in the first-order theory of rewriting expresses the termination property:²

$$\forall t \text{ FIN}_{\rightarrow^+}(t) \wedge \neg \exists u (u \rightarrow^+ u)$$

² The formula characterizes termination of all rewrite systems \mathcal{R} with the property that the induced rewrite relation $\rightarrow_{\mathcal{R}}$ is *finitely branching*.

The predicate $\text{FIN}_{\rightarrow+}$ holds for $t \in \mathcal{T}(\mathcal{F})$ if there are only finitely many terms $u \in \mathcal{T}(\mathcal{F})$ such that $t \rightarrow^+ u$. We consider its complement as it leads to smaller automata:

$$\neg \exists t (\text{INF}_{\rightarrow+}(t) \vee t \rightarrow^+ t)$$

with $\text{INF}_{\rightarrow+} = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow_{\mathcal{R}}^+ u \text{ for infinitely many terms } u \in \mathcal{T}(\mathcal{F})\}$.

Definition 1. Let \circ be an arbitrary binary relation on $\mathcal{T}(\mathcal{F})$. We write INF_{\circ} for the set $\{t \in \mathcal{T}(\mathcal{F}) \mid (t, u) \in \circ \text{ for infinitely many terms } u \in \mathcal{T}(\mathcal{F})\}$.

In [8] the construction of a tree automaton that accepts FIN_{\circ} for an arbitrary RR_2 relation \circ^3 is given. In [16, Appendix A] a correctness proof of the construction is presented, which contains a serious mistake (reported at the end of this section). In this section we give a rigorous and formalized proof of the regularity of INF_{\circ} for arbitrary RR_2 relations \circ .

Theorem 1. The set INF_{\circ} is regular for every RR_2 relation $\circ \subseteq \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$.

The following definition originates from [8].

Definition 2. Given a tree automaton $\mathcal{A} = (\mathcal{F}^{(2)}, Q, Q_f, \Delta)$, the set $Q_{\infty} \subseteq Q$ consists of all states $q \in Q$ such that $\langle \perp, t \rangle \rightarrow_{\Delta}^* q$ for infinitely many terms $t \in \mathcal{T}(\mathcal{F})$.

Example 1. Consider the binary relation

$$\circ = \{(f(a, g^n(b)), g^m(f(a, b))) \mid n = 2 \text{ and } m \geq 1 \text{ or } n \geq 3 \text{ and } m = 1\}$$

The encoding of \circ is accepted by the RR_2 automaton $\mathcal{A} = (\mathcal{F}^{(2)}, Q, Q_f, \Delta)$ with $\mathcal{F} = \{a, b, f, g\}$, $Q = \{0, \dots, 11\}$, $Q_f = \{0\}$, and Δ consisting of the following transition rules:

$$\begin{array}{llll} fg(1, 2) \rightarrow 0 & \perp f(3, 4) \rightarrow 5 & g\perp(6) \rightarrow 2 & b\perp \rightarrow 7 \\ fg(8, 9) \rightarrow 0 & \perp g(5) \rightarrow 5 & g\perp(7) \rightarrow 6 & b\perp \rightarrow 11 \\ af(3, 4) \rightarrow 1 & \perp a \rightarrow 3 & g\perp(10) \rightarrow 9 & ag(5) \rightarrow 1 \\ af(3, 4) \rightarrow 8 & \perp b \rightarrow 4 & g\perp(11) \rightarrow 10 & g\perp(11) \rightarrow 11 \end{array}$$

For instance,

$$\begin{aligned} \langle f(a, g(g(b))), g(f(a, b)) \rangle &= fg(af(\perp a, \perp b), g\perp(g\perp(b\perp))) \\ &\rightarrow_{\Delta}^* fg(af(3, 4), g\perp(g\perp(7))) \rightarrow_{\Delta}^* fg(1, g\perp(6)) \rightarrow_{\Delta} fg(1, 2) \rightarrow_{\Delta} 0 \end{aligned}$$

but $\langle f(a, g(b)), f(a, b) \rangle = ff(aa, gb(b\perp))$ is not accepted.

We have $Q_{\infty} = \{5\}$. State 5 is reached by $\langle \perp, g^n(f(a, b)) \rangle$ for all $n \geq 0$.

³ The relation $\rightarrow_{\mathcal{R}}^+$ is an RR_2 relation for left-linear, right-ground TRSs \mathcal{R} . Other uses of FIN (INF) can be found in [16].

Definition 3. \checkmark Given a tree automaton $\mathcal{A} = (\mathcal{F}^{(2)}, Q, Q_f, \Delta)$, we define the tree automaton $\mathcal{A}_\infty = (\mathcal{F}^{(2)}, Q \cup \bar{Q}, \bar{Q}_f, \Delta \cup \bar{\Delta})$. Here \bar{Q} is a copy of Q where every state is dashed: $\bar{q} \in \bar{Q}$ if and only if $q \in Q$. For every transition rule $fg(q_1, \dots, q_n) \rightarrow q \in \Delta$ we have the following transition rules in $\bar{\Delta}$:

$$fg(q_1, \dots, q_n) \rightarrow \bar{q} \quad \text{if } q \in Q_\infty \text{ and } f = \perp \quad (1)$$

$$fg(q_1, \dots, q_{i-1}, \bar{q}_i, q_{i+1}, \dots, q_n) \rightarrow \bar{q} \quad \text{for all } 1 \leq i \leq n \quad (2)$$

Moreover, for every ϵ -transition $p \rightarrow q \in \Delta$ we add

$$\bar{p} \rightarrow \bar{q} \quad (3)$$

to $\bar{\Delta}$. We write Δ' for $\Delta \cup \bar{\Delta}$.

Dashed states are created by rules of shape (1) and propagated by rules of shapes (2) and (3). The above construction differs from the one in [8]; instead of (1) the latter contains $fg(q_1, \dots, q_n) \rightarrow \bar{q}$ if $q_i \in Q_\infty$ for some $i > \text{arity}(f)$. In an implementation, rather than adding all dashed states and all transition rules of shape (2), the necessary rules would be computed by propagating the dashes created by (1) in order to avoid the appearance of unreachable dashed states. When \mathcal{A}_∞ is used in isolation, a single bit suffices to record that a dashed state occurred during a computation.

Example 2. For the tree automaton \mathcal{A} from Example 1 we obtain \mathcal{A}_∞ by adding the following transition rules (the missing rules of shape (2) involve unreachable states):

$$\perp f(3, 4) \rightarrow \bar{5} \quad \perp g(5) \rightarrow \bar{5} \quad \perp g(\bar{5}) \rightarrow \bar{5} \quad ag(\bar{5}) \rightarrow \bar{1} \quad fg(\bar{1}, 2) \rightarrow \bar{0}$$

The unique final state of \mathcal{A}_∞ is $\bar{0}$. We have $\langle f(a, g(g(b))), g(f(a, b)) \rangle \in L(\mathcal{A}_\infty)$ but there is no term u such that $\langle f(a, g(b)), u \rangle \in L(\mathcal{A}_\infty)$.

The following preliminary lemma is proved by a straightforward induction argument.

Lemma 2. If $t \rightarrow_{\mathcal{A}}^* p$ then $t \rightarrow_{\mathcal{A}_\infty}^* p$. If $C[p] \rightarrow_{\mathcal{A}}^* q$ then $C[p] \rightarrow_{\mathcal{A}_\infty}^* q$ and $C[\bar{p}] \rightarrow_{\mathcal{A}_\infty}^* \bar{q}$. $\checkmark \checkmark$

Theorem 2. Suppose \circ is accepted by the RR_2 automaton \mathcal{A} . If $t \in \text{INF}_\circ$ then $\langle t, u \rangle \in L(\mathcal{A}_\infty)$ for some term $u \in \mathcal{T}(\mathcal{F})$.

Proof. From $t \in \text{INF}_\circ$ and $\circ = L(\mathcal{A})$ we obtain $\langle t, u \rangle \in L(\mathcal{A})$ for infinitely many terms $u \in \mathcal{T}(\mathcal{F})$. Since the signature is finite, there are only finitely many ground terms of any given height. Moreover, $\text{height}(\langle t, u \rangle) = \max(\text{height}(t), \text{height}(u))$. Hence there must exist a term $u \in \mathcal{T}(\mathcal{F})$ with $\langle t, u \rangle \in L(\mathcal{A})$ such that

$$\text{height}(t) + |Q| + 1 < \text{height}(u)$$

This is only possible if there are positions p and q such that $p \notin \mathcal{Pos}(t)$, $pq \in \mathcal{Pos}(u)$, and $|Q| < |q|$. From $\mathcal{Pos}(\langle t, u \rangle) = \mathcal{Pos}(t) \cup \mathcal{Pos}(u)$ we obtain $\langle t, u \rangle|_p = \langle \perp, u|_p \rangle$. Since $\langle t, u \rangle \in L(\mathcal{A})$ there exist states $r \in Q$ and $q \in Q_f$ such that

$$\langle t, u \rangle = \langle t, u \rangle[\langle \perp, u|_p \rangle]_p \rightarrow_{\mathcal{A}}^* \langle t, u \rangle[r]_p \rightarrow_{\mathcal{A}}^* q_f$$

where we assume without loss of generality that the last step in the subsequence $\langle \perp, u|_p \rangle \rightarrow_{\mathcal{A}}^* r$ uses a non-epsilon transition rule.

From $|Q| < |q|$ and $pq \in \mathcal{Pos}(u)$ we infer $|Q| < \text{height}(\langle \perp, u|_p \rangle)$. Hence we can use the pumping lemma (Lemma 1) to conclude the existence of infinitely many terms $v \in \mathcal{T}(\mathcal{F})$ such that $\langle \perp, v \rangle \rightarrow_{\mathcal{A}}^* r$. Hence $r \in Q_\infty$ by Definition 2. Since the last step $\langle \perp, u|_p \rangle \rightarrow_{\mathcal{A}}^* r$ uses a non-epsilon transition rule, we obtain $\langle \perp, u|_p \rangle \rightarrow_{\mathcal{A}_\infty}^* \bar{r}$ using Lemma 2 and a final application of a rule of shape (1). Also using Lemma 2 we obtain $\langle t, u \rangle[\bar{r}]_p \rightarrow_{\mathcal{A}_\infty}^* \bar{q}_f$ as $\langle t, u \rangle[r]_p \rightarrow_{\mathcal{A}}^* q_f$. We conclude $\langle t, u \rangle \in L(\mathcal{A}_\infty)$ as desired. \checkmark

For the reverse direction of Theorem 3 we need two auxiliary results. The first result is proved by a straightforward induction argument. Here the mapping $\varphi: \mathcal{T}(\mathcal{F}^{(2)} \cup Q \cup \bar{Q}) \rightarrow \mathcal{T}(\mathcal{F}^{(2)} \cup Q)$ erases all dashes from states.

Lemma 3. *If $t \in \mathcal{T}(\mathcal{F}^{(2)} \cup Q \cup \bar{Q})$ and $t \rightarrow_{\mathcal{A}_\infty}^* p$ then $\varphi(t) \rightarrow_{\mathcal{A}}^* \varphi(p)$.* \checkmark

With a little bit more effort, we obtain the second auxiliary result. The key step in the proof is identifying the rule of shape (1) that is used to create the first dashed state.

Lemma 4. *If $t \in \mathcal{T}(\mathcal{F}^{(2)})$ and $t \rightarrow_{\mathcal{A}_\infty}^* \bar{p}$ then there exist a state $q \in Q_\infty$, a context C , and a term s such that $C[s] = t$, $\text{root}(s) = \perp f$ for some $f \in \mathcal{F}$, $s \rightarrow_{\mathcal{A}_\infty}^* \bar{q}$, and $C[\bar{q}] \rightarrow_{\mathcal{A}_\infty}^* \bar{p}$.* \checkmark

Theorem 3. *Suppose \circ is accepted by the RR_2 automaton \mathcal{A} . If $\langle t, u \rangle \in L(\mathcal{A}_\infty)$ for some term $u \in \mathcal{T}(\mathcal{F})$ then $t \in \text{INF}_\circ$.*

Proof. From $\langle t, u \rangle \in L(\mathcal{A}_\infty)$ we obtain a final state $\bar{q}_f \in \bar{Q}$ with $\langle t, u \rangle \rightarrow_{\mathcal{A}_\infty}^* \bar{q}_f$. Using Lemma 4, we obtain a context C , a term s with $\text{root}(s) = \perp f$ for some $f \in \mathcal{F}$, and a state $q \in Q_\infty$ such that $C[s] = \langle t, u \rangle$, $s \rightarrow_{\mathcal{A}_\infty}^* \bar{q}$, and $C[\bar{q}] \rightarrow_{\mathcal{A}_\infty}^* \bar{q}_f$. Let p be the position of the hole in C . From $C[s] = \langle t, u \rangle$ and $\text{root}(s) = \perp f$, we infer $p \in \mathcal{Pos}(u) \setminus \mathcal{Pos}(t)$. Since $q \in Q_\infty$ the set $\{v \in \mathcal{T}(\mathcal{F}) \mid \langle \perp, v \rangle \rightarrow_{\mathcal{A}}^* q\}$ is infinite. Hence the set $S = \{u[v]_p \in \mathcal{T}(\mathcal{F}) \mid \langle \perp, v \rangle \rightarrow_{\mathcal{A}}^* q\}$ is infinite, too. Let $u[w]_p \in S$. So $\langle \perp, w \rangle \rightarrow_{\mathcal{A}}^* q$. Since C is ground and $C[\bar{q}] \rightarrow_{\mathcal{A}_\infty}^* \bar{q}_f$, we obtain $C[q] \rightarrow_{\mathcal{A}}^* q_f$ from Lemma 3. We have $C[w] = \langle t, u[w]_p \rangle$ as $p \in \mathcal{Pos}(u) \setminus \mathcal{Pos}(t)$. It follows that $\langle t, u[w]_p \rangle \in L(\mathcal{A})$ and thus there are infinitely many terms u such that $\langle t, u \rangle \in L(\mathcal{A})$. Since $\circ = L(\mathcal{A})$ we conclude the desired $t \in \text{INF}_\circ$. \checkmark

The final step to conclude that the infinity predicate is indeed regular is now easy.

Proof (of Theorem 1). Combining Theorem 2 and Theorem 3 yields the following equivalence:

$$t \in \text{INF}_\circ \iff \langle t, u \rangle \in L(\mathcal{A}_\infty) \text{ for some term } u$$

Hence a tree automaton that accepts INF_\circ is obtained by subjecting \mathcal{A}_∞ to a projection operation (on the first argument). \square

Projection on RR_n automata has been formalized in Isabelle/HOL as part of [10]. \checkmark

The mistake in the proof given in the appendix of [16] is quoted below and corresponds to the proof of Theorem 2:

The set $\mathcal{U} = \{u \in \mathcal{T}(\mathcal{F}) \mid (t, u) \in \circ\}$ is infinite. Since the signature \mathcal{F} is finite, infinitely many terms u in \mathcal{U} have a height greater than t . Hence there exists a position $p \notin \text{Pos}(t)$ such that the set $\mathcal{U}' = \{u \in \mathcal{U} \mid p \in \text{Pos}(u)\}$ is infinite. For every $u \in \mathcal{U}'$ we have $\langle t, u \rangle|_p = \langle \perp, u|_p \rangle$. Since $\langle t, u \rangle$ is accepted by \mathcal{A} and Q is finite, there must exist a state q' such that $\langle \perp, u|_p \rangle \rightarrow_{\mathcal{A}}^* q'$ for infinitely many terms $u \in \mathcal{U}'$. Therefore $q' \in Q_\infty$.

The following example refutes the above reasoning, which is the key step in the proof in [16]. It was found in attempt to formalize the proof.

Example 3. Let $t = f(a, b)$ and consider the infinite set $\mathcal{U} = \{f(f(a, b), g^n(b)) \mid n \geq 1\}$. The automaton

$$\mathcal{A} = (\{f, g, a, b\}^{(2)}, \{q_1, \dots, q_6\}, q_6, \Delta)$$

with Δ consisting of the transition rules

$$\begin{array}{llll} ff(q_4, q_5) \rightarrow q_6 & \perp a \rightarrow q_2 & bg(q_1) \rightarrow q_5 & \perp b \rightarrow q_1 \\ af(q_2, q_3) \rightarrow q_4 & \perp b \rightarrow q_3 & \perp g(q_1) \rightarrow q_1 & \end{array}$$

accepts the relation $\circ = \{t\} \times \mathcal{U}$. Consider the position $p = 11$. We have $p \notin \text{Pos}(t)$ and $p \in \text{Pos}(u)$ for all terms $u \in \mathcal{U}$. Hence $\mathcal{U}' = \mathcal{U}$. Moreover, $\langle t, u \rangle|_p = \langle \perp, a \rangle = \perp a$ for all terms $u \in \mathcal{U}'$. The only state reachable from $\perp a$ is q_2 and clearly $q_2 \notin Q_\infty$.

4 Executable Infinity Predicate

Owing to the definition of Q_∞ , the automaton \mathcal{A}_∞ defined in Definition 3 is not executable. In this section we give an equivalent but executable definition of Q_∞ , which we name Q_∞^e :

$$Q_\infty^e = \{q \mid p \rightsquigarrow p \text{ and } p \rightsquigarrow q \text{ for some state } p \in Q\} \quad \checkmark$$

Here the relation \rightsquigarrow is defined using the inference rules in Figure 1. Before proving that the two definitions are equivalent, we illustrate the definition of Q_∞^e by revisiting Example 1.

$$\frac{\perp f(p_1, \dots, p_n) \rightarrow_{\Delta} p}{p_1 \rightsquigarrow p \quad \dots \quad p_n \rightsquigarrow p} \qquad \frac{p \rightsquigarrow q \quad q \rightarrow_{\Delta} r}{p \rightsquigarrow r} \qquad \frac{p \rightsquigarrow q \quad q \rightsquigarrow r}{p \rightsquigarrow r}$$

Fig. 1. Inference rules for computing Q_{∞}^e .

Example 4. We obtain $3 \rightsquigarrow 5$ and $4 \rightsquigarrow 5$ by applying the first inference rule to the transition rule $\perp f(3, 4) \rightarrow 5$. Similarly, $\perp g(5) \rightarrow 5$ gives rise to $5 \rightsquigarrow 5$. Since \mathcal{A} has no epsilon transitions, no further inferences can be made. It follows that $Q_{\infty}^e = \{5\}$.

We call a term in $\mathcal{T}(\{\perp\} \times \mathcal{F})$ *right-only*. A term in $\mathcal{T}((\{\perp\} \times \mathcal{F}) \cup \{\square\})$ with exactly one occurrence of the hole \square is a right-only context.

Definition 4. We denote the composition of $\rightarrow_{\Delta^{-\epsilon}}$ and $\rightarrow_{\Delta^{\epsilon}}^*$ by \rightarrow_{Δ} .

The proof of the next lemma is straightforward. Note that the relations \rightarrow_{Δ}^* and \rightarrow_{Δ}^* do not coincide on *mixed* terms, involving function symbols and states.

Lemma 5. Let C be a ground context. We have $C[p] \rightarrow_{\Delta}^* q$ if and only if $p \rightarrow_{\Delta}^* p'$ and $C[p'] \rightarrow_{\Delta}^* q$ for some state p' . ☑

Lemma 6. $Q_{\infty} \subseteq Q_{\infty}^e$

Proof. We start by proving the following claim:

$$\text{if } C[p] \rightarrow_{\Delta}^* q \text{ and } C \text{ is a non-empty right-only context then } p \rightsquigarrow q \quad (*)$$

We use induction on the structure of C . If $C = \square$ there is nothing to show. Suppose $C = \perp f(t_1, \dots, C', \dots, t_n)$ where C' is the i -th subterm of C . The sequence $C[p] \rightarrow_{\Delta}^* q$ can be rearranged as $C[p] = \perp f(t_1, \dots, C'[p], \dots, t_n) \rightarrow_{\Delta}^* \perp f(q_1, \dots, q_n) \rightarrow_{\Delta} q' \rightarrow_{\Delta}^* q$. We obtain $q_i \rightsquigarrow q'$ and subsequently $q_i \rightsquigarrow q$ by using the inference rules in Figure 1. If $C' = \square$ then $p = q_i$ and if $C' \neq \square$ then the induction hypothesis yields $p \rightsquigarrow q_i$ and thus $p \rightsquigarrow q$ by transitivity. This concludes the proof of $(*)$. ☑

Assume $q \in Q_{\infty}$, so there exist infinitely many terms t such that $\langle \perp, t \rangle \rightarrow_{\Delta}^* q$. Since the signature is finite, there exist terms of arbitrary height. Thus there exists an arbitrary but fixed term t such that the height of t is greater than the number of states of Q . Write $t = f(t_1, \dots, t_n)$. Since the height of t is greater than the number of the states in Q , there exist a subterm s of t , a state p , and contexts C_1 and $C_2 \neq \square$ such that

1. $\langle \perp, t \rangle = C_1[C_2[\langle \perp, s \rangle]]$,
2. $\langle \perp, s \rangle \rightarrow_{\Delta}^* p$,
3. $C_2[p] \rightarrow_{\Delta}^* p$, and

4. $C_1[p] \rightarrow_{\Delta}^* q$.

From Lemma 5 we obtain a state q' such that $p \rightarrow_{\Delta}^* q'$ and $C_2[q'] \rightarrow_{\Delta}^* p$. Hence $q' \rightsquigarrow p$ by (*). We obtain $q' \rightsquigarrow q'$ from $q' \rightsquigarrow p$ in connection with the inference rule for epsilon transitions. We perform a case analysis of the context C_1 .

- If $C_1 = \square$ then $p \rightarrow_{\Delta}^* q$ and thus $q' \rightsquigarrow q$ follows from $q' \rightsquigarrow p$ in connection with the inference rule for epsilon transitions. Hence $q \in Q_{\infty}^e$.
- If $C_1 \neq \square$ then Lemma 5 yields a state q'' such that $p' \rightarrow_{\Delta}^* q''$ and $C_1[q''] \rightarrow_{\Delta}^* q$. Hence $q'' \rightsquigarrow q$ by (*). We also have $C_2[q'] \rightarrow_{\Delta}^* q''$ and thus $q' \rightsquigarrow q''$ by (*). We obtain $q' \rightsquigarrow q$ from the transitivity rule. Hence also in this case we obtain $q \in Q_{\infty}^e$. ☑

For the following lemma, we need the fact that \mathcal{A} can be assumed to be trim, so every state is productive and reachable. We may do so because Theorem 1 talks about regular relations, and any automaton that accepts the same language as \mathcal{A} will witness the fact that the given relation \circ is regular.

Lemma 7. $Q_{\infty}^e \subseteq Q_{\infty}$, provided that \mathcal{A} is trim.

Proof. In connection with the fact that \mathcal{A} accepts $\circ \subseteq \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$, trimness of \mathcal{A} entails that any run $t \rightarrow_{\Delta}^* q$ is embedded into an accepting run $C[t] \rightarrow_{\Delta}^* C[q] \rightarrow_{\Delta}^* q_f \in Q_f$. So $C[t] = \langle u, v \rangle$ for some $(u, v) \in \circ$, and hence t must be a well-formed term. Moreover, if $\text{root}(t) = \perp f$ for some $f \in \mathcal{F}$ then $t = \langle \perp, u \rangle$ for some term $u \in \mathcal{T}(\mathcal{F})$. We now show the converse of claim (*) in the proof of Lemma 6 for the relation \rightarrow_{Δ}^* :

if $p \rightsquigarrow q$ then $C[p] \rightarrow_{\Delta}^* q$ for some ground right-only context $C \neq \square$ (**)

We prove the claim by induction on the derivation of $p \rightsquigarrow q$. First suppose $p \rightsquigarrow q$ is derived from the transition rule $\perp f(p_1, \dots, p_i, \dots, p_n) \rightarrow q$ in Δ with $p_i = p$. Because all states are reachable by well-formed terms, there exist terms $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F})$ such that $\langle \perp, t \rangle \rightarrow_{\Delta}^* p_i$ for all $1 \leq i \leq n$. Let $C_1 = \perp f(\langle \perp, t_1 \rangle, \dots, \square, \dots, \langle \perp, t_n \rangle)$ where the hole is the i -th argument. We have $C_1[p] \rightarrow_{\Delta}^* \perp f(p_1, \dots, p_i, \dots, p_n) \rightarrow_{\Delta} q$. Next suppose $p \rightsquigarrow q$ is derived from $p \rightsquigarrow q'$ and $q' \rightarrow_{\Delta} q$. The induction hypothesis yields a ground right-only context $C \neq \square$ such that $C[p] \rightarrow_{\Delta}^* q'$. Hence also $C[p] \rightarrow_{\Delta}^* q$. Finally, suppose $p \rightsquigarrow q$ is derived from $p \rightsquigarrow r$ and $r \rightsquigarrow q$. The induction hypothesis yields non-empty ground right-only contexts C_1 and C_2 such that $C_1[p] \rightarrow_{\Delta}^* r$ and $C_2[r] \rightarrow_{\Delta}^* q$. Hence $C[p] \rightarrow_{\Delta}^* q$ for the context $C = C_2[C_1]$. This concludes the proof of (**). ☑

Now let $q \in Q_{\infty}^e$. So there exists a state p such that $p \rightsquigarrow p$ and $p \rightsquigarrow q$. Using (**), we obtain non-empty ground right-only contexts C_1 and C_2 such that $C_1[p] \rightarrow_{\Delta}^* p$ and $C_2[p] \rightarrow_{\Delta}^* q$. Since all states are reachable, there exists a ground term $t \in \mathcal{T}(\mathcal{F}^{(2)})$ such that $t \rightarrow_{\Delta}^* p$. Hence $C_2[t] \rightarrow_{\Delta}^* q$ and, by the observation made at the beginning of the proof, $C_2[t]$ is a well-formed term. Since C_2 is right-only, it follows that $t = \langle \perp, u \rangle$ for some term $u \in \mathcal{T}(\mathcal{F})$. Now consider the infinitely many terms $t_n = C_2[C_1^n[t]]$ for $n \geq 0$. We have $t_n \rightarrow_{\Delta}^* q$ and t_n is right-only by construction. Hence $q \in Q_{\infty}$. ☑

Corollary 1. $Q_\infty^e = Q_\infty$, provided that \mathcal{A} is trim. \square

5 Normal Form Predicate

The normal form predicate NF can be defined in the first-order theory of rewriting as

$$\text{NF}(t) \iff \neg \exists u (t \rightarrow u)$$

and this gives rise to the following procedure:

1. An RR_2 automaton is constructed that accepts the encoding of the rewrite relation \rightarrow .
2. The RR_2 automaton of step 1 is *projected* into a tree automaton that accepts the set of reducible ground terms.
3. Complementation is applied to the automaton of step 2 to obtain a tree automaton that accepts the set of ground normal forms.

Since projection may transform a deterministic tree automaton into a non-deterministic one, this is inefficient. In this section we provide a direct construction of a tree automaton that accepts the set of ground normal forms of a left-linear TRS, which goes back to Comon [5], and present a formalized correctness proof. Throughout this section \mathcal{R} is assumed to be left-linear.

We start with defining some preliminary concepts.

Definition 5. Given a signature \mathcal{F} , we write \mathcal{F}_\perp for the extension of \mathcal{F} with a fresh constant symbol \perp . Given $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, t^\perp denotes the result of replacing all variables in t by \perp :

$$x^\perp = \perp \qquad f(t_1, \dots, t_n)^\perp = f(t_1^\perp, \dots, t_n^\perp) \quad \checkmark$$

We define the partial order \leq on $\mathcal{T}(\mathcal{F}_\perp)$ as the least congruence that satisfies $\perp \leq t$ for all terms $t \in \mathcal{T}(\mathcal{F}_\perp)$:

$$\frac{}{\perp \leq t} \qquad \frac{t_1 \leq u_1 \quad \dots \quad t_n \leq u_n}{f(t_1, \dots, t_n) \leq f(u_1, \dots, u_n)} \quad \checkmark$$

The partial map $\uparrow: \mathcal{T}(\mathcal{F}_\perp) \times \mathcal{T}(\mathcal{F}_\perp) \rightarrow \mathcal{T}(\mathcal{F}_\perp)$ is defined as follows:

$$\perp \uparrow t = t \uparrow \perp = t \quad f(t_1, \dots, t_n) \uparrow f(u_1, \dots, u_n) = f(t_1 \uparrow u_1, \dots, t_n \uparrow u_n) \quad \checkmark$$

It is not difficult to show that $t \uparrow u$ is the least upper bound of comparable terms t and u .

Definition 6. \checkmark Let \mathcal{R} be a TRS over a signature \mathcal{F} . We write T^\perp for the set $\{t^\perp \mid t \triangleleft \ell \rightarrow r \in \mathcal{R}\} \cup \{\perp\}$. The set T_\uparrow is obtained by closing T^\perp under \uparrow .

Example 5. Consider the TRS \mathcal{R} consisting of following rules:

$$h(f(g(a), x, y)) \rightarrow g(a) \quad g(f(x, h(x), y)) \rightarrow x \quad h(f(x, y, h(a))) \rightarrow h(x)$$

We start by collecting the subterms of the left-hand sides:

$$T^\perp = \{\perp, a, g(a), h(\perp), h(a), f(g(a), \perp, \perp), f(\perp, h(\perp), \perp), f(\perp, \perp, h(a))\}$$

Closing T^\perp under \uparrow adds the following terms:

$$\begin{aligned} f(g(a), \perp, \perp) \uparrow f(\perp, h(\perp), \perp) &= f(g(a), h(\perp), \perp) \\ f(\perp, \perp, h(a)) \uparrow f(\perp, h(\perp), \perp) &= f(\perp, h(\perp), h(a)) \\ f(g(a), h(\perp), \perp) \uparrow f(\perp, h(\perp), h(a)) &= f(g(a), h(\perp), h(a)) \end{aligned}$$

Lemma 8. *The set T_\uparrow is finite.*

Proof. If $t \uparrow u$ is defined then $\mathcal{Pos}(t \uparrow u) = \mathcal{Pos}(t) \cup \mathcal{Pos}(u)$. It follows that the positions of terms in $T_\uparrow \setminus T^\perp$ are positions of terms in T^\perp . Since T^\perp is finite, there are only finitely many such positions. Hence the finiteness of T_\uparrow follows from the finiteness of \mathcal{F} . \square

Although the above proof is simple enough, we formalized the proof below which is based on a concrete algorithm to compute T_\uparrow . Actually, the algorithm presented below is based on a general saturation procedure, which is of independent interest.

Definition 7. *Let $f: U \times U \rightarrow U$ be a (possibly partial) function and let S be a finite subset of U . The closure $C_f(S)$ is the least extension of S with the property that $f(a, b) \in C_f(S)$ whenever $a, b \in C_f(S)$ and $f(a, b)$ is defined.*

The following lemma provides a sufficient condition for closures to exist. The proof gives a concrete algorithm to compute the closure.

Lemma 9. *If f is a total, associative, commutative, and idempotent function then $C_f(S)$ exists and is finite.*

Proof. A straightforward induction proof reveals that for every $a \in C_f(S)$ there exist elements $a_1, \dots, a_n \in S$ such that $a = f(a_1, f(a_2, \dots, f(a_{n-1}, a_n) \dots))$. Select an arbitrary element $b \in S$. If b is among a_1, \dots, a_n then, using the properties of f , we obtain $a \in \{f(b, c) \mid c \in C_f(S \setminus \{b\})\}$. If b is not among a_1, \dots, a_n then $a \in C_f(S \setminus \{b\})$. Hence

$$C_f(S) = C_f(S \setminus \{b\}) \cup \{b\} \cup \{f(b, c) \mid c \in C_f(S \setminus \{b\})\}$$

for every $b \in S$. Since S is finite, this gives rise to an iterative algorithm to compute $C_f(S)$, which is given in Listing 5. In each iteration only finitely many elements are added. Hence $C_f(S)$ is finite. \square

```

saturate(S):
  I ← ∅
  for all x ∈ S do
    I ← {x} ∪ I ∪ {f(x, y) | y ∈ I}
  return I

```

Listing 1. Iterative closure algorithm.

Since our function \uparrow is partial, we need to lift it to a total function that preserves associativity and commutativity. In our abstract setting this entails finding a binary predicate P on U such that $f(a, b)$ is defined if $P(a, b)$ holds. In addition, the following properties need to be fulfilled:

- P is reflexive and symmetric,
- if $P(a, f(b, c))$ and $P(b, c)$ hold then $P(a, b)$ and $P(f(a, b), c)$ hold as well, for all $a, b, c \in U$.

For the details we refer to the formalization. ✔✔✔✔

Definition 8. ✔ *The tree automaton $\mathcal{A}_{\text{NF}(\mathcal{R})} = (\mathcal{F}, Q, Q_f, \Delta)$ is defined as follows: $Q = Q_f = T_{\uparrow}$ and Δ consists of all transition rules*

$$f(p_1, \dots, p_n) \rightarrow q \quad \text{✔}$$

such that $f(p_1, \dots, p_n)$ is no redex and q is the maximal element of Q satisfying $q \leq f(p_1, \dots, p_n)$.⁴

Example 6. For the TRS \mathcal{R} of Example 5, the tree automaton $\mathcal{A}_{\text{NF}(\mathcal{R})}$ consists of the following transition rules:

$$\begin{array}{l}
\mathbf{a} \rightarrow 1 \quad \mathbf{g}(p) \rightarrow \begin{cases} 2 & \text{if } p = 1 \\ 0 & \text{if } p \notin \{1, 6, 9, 10\} \end{cases} \quad \mathbf{h}(p) \rightarrow \begin{cases} 4 & \text{if } p = 1 \\ 3 & \text{if } p \notin \{1, 8, 10\} \end{cases} \\
\mathbf{f}(p, q, r) \rightarrow \begin{cases} 5 & \text{if } p = 2, q \notin \{3, 4\} \\ 6 & \text{if } p \neq 2, q \in \{3, 4\}, r \neq 4 \\ 7 & \text{if } q \notin \{3, 4\}, r = 4 \\ 8 & \text{if } p = 2, q \in \{3, 4\}, r \neq 4 \\ 9 & \text{if } p \neq 2, q \in \{3, 4\}, r = 4 \\ 10 & \text{if } p = 2, q \in \{3, 4\}, r = 4 \\ 0 & \text{otherwise} \end{cases}
\end{array}$$

Here we use the following abbreviations:

$$\begin{array}{llll}
0 = \perp & 3 = \mathbf{h}(\perp) & 6 = \mathbf{f}(\perp, \mathbf{h}(\perp), \perp) & 8 = \mathbf{f}(\mathbf{g}(\mathbf{a}), \mathbf{h}(\perp), \perp) \\
1 = \mathbf{a} & 4 = \mathbf{h}(\mathbf{a}) & 7 = \mathbf{f}(\perp, \perp, \mathbf{h}(\mathbf{a})) & 9 = \mathbf{f}(\perp, \mathbf{h}(\perp), \mathbf{h}(\mathbf{a})) \\
2 = \mathbf{g}(\mathbf{a}) & 5 = \mathbf{f}(\mathbf{g}(\mathbf{a}), \perp, \perp) & & 10 = \mathbf{f}(\mathbf{g}(\mathbf{a}), \mathbf{h}(\perp), \mathbf{h}(\mathbf{a}))
\end{array}$$

⁴ Since states are terms from T_{∞} here, Definition 5 applies.

As can be seen from the above example, the tree automaton $\mathcal{A}_{\text{NF}(\mathcal{R})}$ is not completely defined. Unlike the construction in [5], we do not have an additional state that is reached by all reducible ground terms.

Before proving that $\mathcal{A}_{\text{NF}(\mathcal{R})}$ accepts the ground normal forms of \mathcal{R} , we first show that $\mathcal{A}_{\text{NF}(\mathcal{R})}$ is well-defined, which amounts to showing that for every $f(p_1, \dots, p_n)$ with $f \in \mathcal{F}$ and $p_1, \dots, p_n \in T_\uparrow$ the set of states q such that $q \leq f(p_1, \dots, p_n)$ has a maximum element with respect to the partial order \leq .

Lemma 10. *For every term $t \in T_\uparrow$ the set $\{s \in T_\uparrow \mid s \leq t\}$ has a unique maximal element.*

Proof. Let $S = \{s \in T_\uparrow \mid s \leq t\}$. Because $\perp \leq t$ and $\perp \in T_\uparrow$, $S \neq \emptyset$. If $s_1, s_2 \in T$ then $s_1 \leq t$ and $s_2 \leq t$ and thus $s_1 \uparrow s_2$ is defined and satisfies $s_1 \uparrow s_2 \leq t$. Since T_\uparrow is closed under \uparrow , $s_1 \uparrow s_2 \in T_\uparrow$ and thus $s_1 \uparrow s_2 \in S$. Consequently, S has a unique maximal element. \square

The next lemma is a trivial consequence of the fact that $\mathcal{A}_{\text{NF}(\mathcal{R})}$ has no epsilon transitions.

Lemma 11. *The tree automaton $\mathcal{A}_{\text{NF}(\mathcal{R})}$ is deterministic.* \checkmark

Lemma 12. *If $t \in \mathcal{T}(\mathcal{F})$ with $t \rightarrow_\Delta^* q$ and $s^\perp \leq t^\perp$ for a proper subterm s of some left-hand side of \mathcal{R} then $s^\perp \leq q$.*

Proof. We use structural induction on t . Let $t = f(t_1, \dots, t_n)$. We have $t \rightarrow_\Delta^* f(q_1, \dots, q_n) \rightarrow_\Delta q$. We proceed by case analysis on s . If s is a variable then $s^\perp = \perp$ and, as \perp is minimal in \leq , we obtain $s^\perp \leq q$. Otherwise we must have $\text{root}(s) = f$ from the assumption $s^\perp \leq t^\perp$. So we may write $s = f(s_1, \dots, s_n)$. The induction hypothesis yields $s_i^\perp \leq q_i$ for all $1 \leq i \leq n$. Hence $s^\perp = f(s_1^\perp, \dots, s_n^\perp) \leq f(q_1, \dots, q_n)$. Additionally we have $s^\perp \in Q$ by Definition 8 as s is a proper subterm of a left-hand side of \mathcal{R} . Since $f(q_1, \dots, q_n) \rightarrow q$ is a transition rule, we obtain $f(s_1, \dots, s_n)^\perp \leq q$ from the maximality of q . \checkmark

Using the previous result we can prove that no redex of \mathcal{R} reaches a state in $\mathcal{A}_{\text{NF}(\mathcal{R})}$.

Lemma 13. *If $t \in \mathcal{T}(\mathcal{F})$ is a redex then $t \rightarrow_\Delta^* q$ for no state $q \in T_\uparrow$.*

Proof. We have $\ell^\perp \leq t$ for some left-hand side ℓ of \mathcal{R} . For a proof by contradiction, assume $t \rightarrow_\Delta^* q$. Write $t = f(t_1, \dots, t_n)$. We have $t \rightarrow_\Delta^* f(q_1, \dots, q_n) \rightarrow_\Delta q$ and obtain $\ell^\perp \leq f(q_1, \dots, q_n)$ by a case analysis on ℓ and Lemma 12. Therefore the transition rule $f(q_1, \dots, q_n) \rightarrow_\Delta q$ cannot exist by Definition 8. \checkmark

Lemma 14. *If $t \rightarrow_\Delta^* q$ and $t \in \mathcal{T}(\mathcal{F})$ then $q \leq t$.*

Proof. We use structural induction on t . Let $t = f(t_1, \dots, t_n)$. We have $t \rightarrow_\Delta^* f(q_1, \dots, q_n) \rightarrow_\Delta^* q$. The induction hypothesis yields $q_i \leq t_i$ for all $1 \leq i \leq n$ and thus also $f(q_1, \dots, q_n) \leq f(t_1, \dots, t_n)$. We have $q \leq f(q_1, \dots, q_n)$ by Definition 8 and thus $q \leq t$ by the transitivity of \leq . \checkmark

Lemma 15. *If $t \in \text{NF}(\mathcal{R})$ then $t \rightarrow_{\Delta}^* q$ for some state $q \in T_{\uparrow}$.*

Proof. We use structural induction on t . Let $t = f(t_1, \dots, t_n)$. Since $t_1, \dots, t_n \in \text{NF}(\mathcal{R})$ we obtain $f(t_1, \dots, t_n) \rightarrow_{\Delta}^* f(q_1, \dots, q_n)$ from the induction hypothesis. Suppose $f(q_1, \dots, q_n)$ is a redex, so $l^{\perp} \leq f(q_1, \dots, q_n)$ for some left-hand side l of \mathcal{R} . From Lemma 14 we obtain $q_i \leq t_i$ for all $1 \leq i \leq n$ and thus $f(q_1, \dots, q_n) \leq f(t_1, \dots, t_n)$. Hence $l^{\perp} \leq f(t_1, \dots, t_n)$. This however contradicts the assumption that t is a normal form. (Here we need left-linearity of \mathcal{R} .) Therefore $f(q_1, \dots, q_n)$ is no redex and thus, using Lemma 10, there exists a transition $f(q_1, \dots, q_n) \rightarrow q$ in Δ and thus $t \rightarrow_{\Delta}^* q$. ✔

Theorem 4. *If \mathcal{R} is a left-linear TRS then $L(\mathcal{A}_{\text{NF}(\mathcal{R})}) = \text{NF}(\mathcal{R})$.*

Proof. Let $t \in \mathcal{T}(\mathcal{F})$. If $t \in \text{NF}(\mathcal{R})$ then $t \rightarrow_{\Delta}^* q$ for some state $q \in T_{\uparrow}$ by Lemma 15. Since all states in T_{\uparrow} are final, $t \in L(\mathcal{A}_{\text{NF}(\mathcal{R})})$. ✔

Next assume $t \notin \text{NF}(\mathcal{R})$. Hence $t = C[s]$ for some redex s . According to Lemma 13 s does not reach a state in $\mathcal{A}_{\text{NF}(\mathcal{R})}$. Hence also t cannot reach a state and thus $t \notin L(\mathcal{A}_{\text{NF}(\mathcal{R})})$. ✔

6 Conclusion and Future Work

In this paper we presented formalized correctness proofs of the regularity of the infinity and normal form predicates in the first-order theory of rewriting. For the former we also provided an executable version, which is important for checking certificates that will be provided in a future version of FORT. Our results are an important step towards the ultimate goal of proving the correctness of the decisions reported by FORT, but much work remains to be done. We are developing a certification language which reflects the high-level proof steps in the decision procedure for the full first-order theory of rewriting. This language will be independent of FORT. In particular, details of the intermediate tree automata computed by FORT will not be part of certificates. This keeps the certificates small and avoids having to implement a verified (and expensive) equivalence check on tree automata. We will provide executable Isabelle code for each of the constructs in the certification language, and so this involves replaying the automata constructions in Isabelle.

We conclude the paper by providing some details of the size of our formalization in Table 1.

Acknowledgments. We thank Bertram Felgenhauer and T. V. H. Prathamash for contributions in the early stages of this work. The comments by the reviewers helped to improve the presentation of the paper.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998). <https://doi.org/10.1017/CBO9781139172752>

Table 1. Formalization data.

	theory	lines	facts	defs
	Saturation.thy	233	22	3
	Tree_Automata_Pumping.thy	371	40	2
	NF.thy	404	40	7
	RR2_Infinite.thy	603	48	5
	RR2_Infinite_Impl.thy	240	14	2
	total	1851	164	19

- Barrett, C., Deters, M., de Moura, L., Oliveras, A., Stump, A.: 6 years of SMT-COMP. *Journal of Automated Reasoning* **50**(3), 243–277 (2013). <https://doi.org/10.1007/s10817-012-9246-5>
- Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). <https://doi.org/10.1007/978-3-662-07964-5>
- Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): *Handbook of Satisfiability*, *Frontiers in Artificial Intelligence and Applications*, vol. 185. IOS Press (2009)
- Comon, H.: Sequentiality, monadic second-order logic and tree automata. *Information and Computation* **157**(1-2), 25–51 (2000). <https://doi.org/10.1006/inco.1999.2838>
- Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: *Tree automata techniques and applications* (2008), <http://tata.gforge.inria.fr/>
- Dauchet, M., Tison, S.: The theory of ground rewrite systems is decidable. In: *Proc. 5th IEEE Symposium on Logic in Computer Science*. pp. 242–248 (1990). <https://doi.org/10.1109/LICS.1990.113750>
- Dauchet, M., Tison, S.: The theory of ground rewrite systems is decidable (extended version). Tech. Rep. I.T. 197, LIFL (1990)
- Endrullis, J., Zantema, H.: Proving non-termination by finite automata. In: Fernández, M. (ed.) *Proc. 26th International Conference on Rewriting Techniques and Applications*. *Leibniz International Proceedings in Informatics*, vol. 36, pp. 160–168 (2015). <https://doi.org/10.4230/LIPIcs.RTA.2015.257>
- Felgenhauer, B., Middeldorp, A., Prathamesh, T.V.H., Rapp, F.: A verified ground confluence tool for linear variable-separated rewrite systems in Isabelle/HOL. In: Mahboubi, A., Myreen, M.O. (eds.) *Proc. 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. pp. 132–143 (2019). <https://doi.org/10.1145/3293880.3294098>
- Felgenhauer, B., Thiemann, R.: Reachability, confluence, and termination analysis with state-compatible automata. *Information and Computation* **253**(3), 467–483 (2017). <https://doi.org/10.1016/j.ic.2016.06.011>
- Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) *Proc. 25th International Conference on Tools and Algorithms for the*

- Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 11429, pp. 156–166 (2019). https://doi.org/10.1007/978-3-030-17502-3_10
13. Middeldorp, A., Nagele, J., Shintani, K.: Confluence competition 2019. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Proc. 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 11429, pp. 25–40 (2019). https://doi.org/10.1007/978-3-030-17502-3_2
 14. Nagaya, T., Toyama, Y.: Decidability for left-linear growing term rewriting systems. *Information and Computation* **178**(2), 499–514 (2002). <https://doi.org/10.1006/inco.2002.3157>
 15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). <https://doi.org/10.1007/3-540-45949-9>
 16. Rapp, F., Middeldorp, A.: Automating the first-order theory of left-linear right-ground term rewrite systems. In: Kesner, D., Pientka, B. (eds.) Proc. 1st International Conference on Formal Structures for Computation and Deduction. *Leibniz International Proceedings in Informatics*, vol. 52, pp. 36:1–36:12 (2016). <https://doi.org/10.4230/LIPIcs.FSCD.2016.36>
 17. Rapp, F., Middeldorp, A.: FORT 2.0. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Proc. 9th International Joint Conference on Automated Reasoning. *LNAI*, vol. 10900, pp. 81–88 (2018). https://doi.org/10.1007/978-3-319-94205-6_6
 18. Terese (ed.): *Term Rewriting Systems*, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
 19. Thiemann, R., Sternagel, C.: Certification of termination proofs using CETA . In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Proc. 22nd International Conference on Theorem Proving in Higher Order Logics. *Lecture Notes in Computer Science*, vol. 5674, pp. 452–468 (2009). https://doi.org/10.1007/978-3-642-03359-9_31

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

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





Fold/Unfold Transformations for Fixpoint Logic

Naoki Kobayashi¹, Grigory Fedyukovich², and Aarti Gupta³

¹ The University of Tokyo, Tokyo, Japan, koba@is.s.u-tokyo.ac.jp

² Florida State University, Tallahassee, USA, grigory@cs.fsu.edu

³ Princeton University, Princeton, USA, aartig@cs.princeton.edu

Abstract. Fixpoint logics have recently been drawing attention as common foundations for automated program verification. We formalize fold/unfold transformations for fixpoint logic formulas and show how they can be used to enhance a recent fixpoint-logic approach to automated program verification, including automated verification of relational and temporal properties. We have implemented the transformations in a tool and confirmed its effectiveness through experiments.

1 Introduction

A wide range of program properties can be verified by reducing to satisfiability/validity in a fixpoint logic [3–6, 18, 20, 22, 23, 29, 35]. In this paper, we build on top of **MuArith**, a first-order logic with least/greatest fixpoint operators and integer arithmetic, recently proposed by Kobayashi et al. [22]. It offers a powerful tool to handle the full class of modal μ -calculus properties of while-programs (imperative programs with loops but without general recursion). In contrast, earlier studies on temporal program verification require different methods for each subclass of the modal μ -calculus properties, such as LTL [12, 16, 28], CTL [2, 3, 13, 34], and CTL* [11]. The recent program verifier based on **MuArith** [22] is effective in practice, i.e., by exploiting general-purpose solvers for Satisfiability Modulo Theories (SMT) and Constrained Horn Clauses (CHC), it can outperform tools designed specifically for CTL verification of C programs [13].

Despite these promising results, the generality of the fixpoint logic approach come at a cost. Since fixpoint logic formulas obtained by reduction from various verification problems often involve nested fixpoint operators, it could be challenging to check the validity of these formulas automatically. To enhance the capability of fixpoint logic provers, in this paper, we propose novel fold/unfold transformations and prove their correctness. These transformations are generally used to simplify relational verification, and in particular, to reduce the number of recurrences used in the program (or a set of programs) under analysis. Originally proposed for logic programming [8, 19, 32], they have been recently adopted for determining the satisfiability of CHC [15, 26] and allow discovery of *relational* invariants for a pair of loopy (or recursive) programs, as opposed to invariants within each individual program. Our transformations can be regarded as extensions of such transformations for a fixpoint logic, where quantifiers and arbitrarily nested least/greatest fixpoint operators are allowed.

We also present a procedure that seeks a way to apply the proposed fold/unfold transformations efficiently. Besides non-determinism in the choice of which fixpoint formulas to unfold, our “fold” operation replaces a formula ϕ with P (where P is the predicate defined by $P \triangleq \phi$) and requires various reasoning to convert the current goal formula to a form $E[\phi]$, where the form of E can be more complex than in the case of fold/unfold transformations for logic programming or CHC.

We have implemented the transformations and integrated them with the program verifier `Mu2CHC` [22] based on `MuArith`. We considered a number of examples of `MuArith` formulas which include formulas obtained from program verification problems for checking relational and temporal properties. Our new transformations allowed `Mu2CHC` to solve these formulas, which would not be doable otherwise.

To sum up, our contributions are: (i) a formalization of fold/unfold transformations for a fixpoint logic and proofs of their soundness, (ii) demonstration of the usefulness of the proposed transformations for verification of relational and temporal properties of programs, and (iii) a concrete procedure for automated transformation and its implementation and experiments.

The rest of this paper is structured as follows. Section 2 reviews the definition of the first-order fixpoint logic `MuArith` [22], and reductions from program verification problems to validity checking in `MuArith`. Section 3 formalizes our transformations and proves their correctness. Section 4 shows applications of our transformations to verification of relational and temporal properties of recursive programs. Section 5 reports an implementation and experimental results. Section 6 discusses related work and Section 7 concludes the paper.

2 First-Order Fixpoint Logic `MuArith`

We review the first-order fixpoint logic `MuArith` [22] in this section. `MuArith` is a variation of Mu-Arithmetic studied by Lubarsky [25] and Bradfield [7], obtained by replacing natural numbers with integers.

2.1 Syntax

The set of (propositional) formulas, ranged over by φ , is defined in the following grammar.

$$\begin{aligned} \varphi \text{ (formulas)} &::= a_1 \geq a_2 \mid P^{(k)}(a_1, \dots, a_k) \mid \\ &\quad \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \forall x. \varphi \mid \exists x. \varphi \\ P^{(k)} \text{ (} k\text{-ary predicates)} &::= X^{(k)} \mid \lambda(x_1, \dots, x_k). \varphi \mid \\ &\quad \mu X^{(k)}(x_1, \dots, x_k). \varphi \mid \nu X^{(k)}(x_1, \dots, x_k). \varphi \\ a \text{ (arithmetic expressions)} &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \end{aligned}$$

The metavariable φ represents a proposition, and P denotes a predicate on (a tuple of) integers. We write \top for $0 \geq 0$ and \perp for $0 \geq 1$. In examples,

we may also use other relational symbols such as $>$ and $=$. The meta-variable x denotes an integer, and the meta-variable $X^{(k)}$ denotes a k -ary first-order predicate variable. We write $\text{ar}(X^{(k)})$ for the arity of the predicate variable $X^{(k)}$, i.e., k ; we often omit the superscript (k) and just write X for a predicate variable. The predicate $\mu X^{(k)}(x_1, \dots, x_k).\varphi$ (resp. $\nu X^{(k)}(x_1, \dots, x_k).\varphi$) denotes the least (resp. greatest) predicate X such that $X(x_1, \dots, x_k)$ equals φ .

Example 1. Let $\mu X(x).(x = 0 \vee X(x - 1))$ denote the least predicate X , such that $X(x) \equiv x = 0 \vee X(x - 1) \equiv x = 0 \vee x - 1 = 0 \vee X(x - 2) \equiv \dots$, i.e., $\lambda(x).x \geq 0$. In contrast, $\nu X(x).(x = 0 \vee X(x - 1))$ denotes $\lambda(x).\top$. \square

We write $\mathbf{FV}(\varphi)$ for the set of free (predicate and integer) variables in φ ; $\forall x, \exists x, \mu X^{(k)}, \nu X^{(k)}$, and λx are binders. We sometimes write \tilde{x} for a sequence of variables x_1, \dots, x_k . We often write $\overline{\varphi}$ and \overline{X} for the De Morgan dual of a formula φ and a predicate variable X , respectively. For example, $\overline{\mu X(x).x = 0 \vee X(x - 1)} = \nu \overline{X}(x).x \neq 0 \wedge \overline{X}(x - 1)$. Here, \overline{X} is a predicate variable, so the righthand side is α -equivalent to $\nu X(x).x \neq 0 \wedge X(x - 1)$. The overline for X is used to indicate that it corresponds to the dual of X in the original formula $\mu X(x).x = 0 \vee X(x - 1)$.

2.2 Semantics

In this subsection, we define the formal semantics of formulas. Let \mathbf{Z} be the set of integers, and $\mathbf{B} = \{\top_{\mathbf{B}}, \perp_{\mathbf{B}}\}$, with $\perp_{\mathbf{B}} \sqsubseteq_{\mathbf{B}} \top_{\mathbf{B}}$. Let \mathbf{D}_k be the set $\mathbf{Z}^k \rightarrow \mathbf{B}$ of functions (where \mathbf{Z}^k denotes the set of tuples consisting of k integers). We define the partial order \sqsubseteq_k on \mathbf{D}_k by:

$$f \sqsubseteq_k g \Leftrightarrow \forall n_1, \dots, n_k \in \mathbf{Z}. f(n_1, \dots, n_k) \sqsubseteq_{\mathbf{B}} g(n_1, \dots, n_k).$$

Note that $(\mathbf{D}_k, \sqsubseteq_k)$ is a complete lattice, with $\lambda x_1. \dots \lambda x_k. \perp_{\mathbf{B}}$ and $\lambda x_1. \dots \lambda x_k. \top_{\mathbf{B}}$ as the least and greatest elements. We write \perp_k and \top_k for $\lambda x_1. \dots \lambda x_k. \perp_{\mathbf{B}}$ and $\lambda x_1. \dots \lambda x_k. \top_{\mathbf{B}}$, respectively. We also write $\sqcap^{(k)}$ (resp., $\sqcup^{(k)}$) for the greatest lower (resp., least upper) bound with respect to \sqsubseteq_k . We often omit k and \mathbf{B} and just write $\top, \perp, \sqsubseteq, \sqcap, \sqcup$, etc.. We often identify \mathbf{B} and $\mathbf{D}_0 = \mathbf{Z}^0 \rightarrow \mathbf{B}$. We write $\mathbf{D}_k \rightarrow \mathbf{D}_\ell$ for the set of monotonic functions from \mathbf{D}_k to \mathbf{D}_ℓ .

We write \mathbf{Env} for the set of functions that map each integer variable to an integer, and each k -ary predicate variable to an element of \mathbf{D}_k . For a formula φ (resp., a predicate P and an expression a) and an environment $\rho \in \mathbf{Env}$ such that $\mathbf{FV}(\varphi) \subseteq \text{dom}(\mathbf{Env})$ (resp., $\mathbf{FV}(P) \subseteq \text{dom}(\mathbf{Env})$ and $\mathbf{FV}(a) \subseteq \text{dom}(\mathbf{Env})$), Fig. 1 defines the semantics $\llbracket \cdot \rrbracket \rho$ of φ (resp., P and a), where for a monotonic function $F \in \mathbf{D}_k \rightarrow \mathbf{D}_k$, $\mathbf{LFP}^{(k)}(F) = \sqcap^{(k)} \{f \in \mathbf{D}_k \mid f \sqsupseteq_k F(f)\}$ and $\mathbf{GFP}^{(k)}(F) = \sqcup^{(k)} \{f \in \mathbf{D}_k \mid f \sqsubseteq_k F(f)\}$. When φ and P are closed (i.e., do not contain free variables), we just write $\llbracket \varphi \rrbracket$ and $\llbracket P \rrbracket$ for $\llbracket \varphi \rrbracket \emptyset$ and $\llbracket P \rrbracket \emptyset$ respectively. By abuse of notation, we often write $\varphi \sqsupseteq \psi$ if $\llbracket \varphi \rrbracket \rho \sqsupseteq \llbracket \psi \rrbracket \rho$ for any (valid) environment ρ such that $\mathbf{FV}(\varphi) \cup \mathbf{FV}(\psi) \subseteq \text{dom}(\rho)$, and $\varphi \equiv \psi$ if $\llbracket \varphi \rrbracket \rho = \llbracket \psi \rrbracket \rho$; similarly for predicates. For example, $\exists z.(x > z \wedge z > y) \equiv (x > y + 1) \sqsupseteq (x > y + 2)$.

$$\begin{aligned}
\llbracket a_1 \geq a_2 \rrbracket \rho &= \begin{cases} \top & \text{if } \llbracket a_1 \rrbracket \rho \geq \llbracket a_2 \rrbracket \rho \\ \perp & \text{if } \llbracket a_1 \rrbracket \rho < \llbracket a_2 \rrbracket \rho \end{cases} \\
\llbracket P(a_1, \dots, a_k) \rrbracket \rho &= \llbracket P \rrbracket \rho(\llbracket a_1 \rrbracket \rho, \dots, \llbracket a_k \rrbracket \rho) \\
\llbracket X \rrbracket \rho &= \rho(X) \\
\llbracket \varphi_1 \vee \varphi_2 \rrbracket \rho &= \llbracket \varphi_1 \rrbracket \rho \sqcup \llbracket \varphi_2 \rrbracket \rho \\
\llbracket \varphi_1 \wedge \varphi_2 \rrbracket \rho &= \llbracket \varphi_1 \rrbracket \rho \sqcap \llbracket \varphi_2 \rrbracket \rho \\
\llbracket \forall x. \varphi \rrbracket \rho &= \bigsqcap_{n \in \mathbf{Z}} \llbracket \varphi \rrbracket \rho \{x \mapsto n\} \\
\llbracket \exists x. \varphi \rrbracket \rho &= \bigsqcup_{n \in \mathbf{Z}} \llbracket \varphi \rrbracket \rho \{x \mapsto n\} \\
\llbracket \lambda(x_1, \dots, x_k). \varphi \rrbracket \rho &= \lambda(n_1, \dots, n_k) \in \mathbf{Z}^k. \llbracket \varphi \rrbracket \rho \{x_1 \mapsto n_1, \dots, x_k \mapsto n_k\} \\
\llbracket \mu X^{(k)}(x_1, \dots, x_k). \varphi \rrbracket \rho &= \\
&\quad \mathbf{LFP}^{(k)}(\lambda f \in \mathbf{D}_k. \lambda(n_1, \dots, n_k). \llbracket \varphi \rrbracket \rho \{X \mapsto f, x_1 \mapsto n_1, \dots, x_k \mapsto n_k\}) \\
\llbracket \nu X^{(k)}(x_1, \dots, x_k). \varphi \rrbracket \rho &= \\
&\quad \mathbf{GFP}^{(k)}(\lambda f \in \mathbf{D}_k. \lambda(n_1, \dots, n_k). \llbracket \varphi \rrbracket \rho \{X \mapsto f, x_1 \mapsto n_1, \dots, x_k \mapsto n_k\}) \\
&\quad \llbracket n \rrbracket \rho = n \\
&\quad \llbracket x \rrbracket \rho = \rho(x) \\
\llbracket a_1 + a_2 \rrbracket \rho &= \llbracket a_1 \rrbracket \rho + \llbracket a_2 \rrbracket \rho \\
\llbracket a_1 - a_2 \rrbracket \rho &= \llbracket a_1 \rrbracket \rho - \llbracket a_2 \rrbracket \rho
\end{aligned}$$

Fig. 1. The semantics of formulas.

Example 2. Recall formula $\mu X(x).x = 0 \vee X(x - 1)$ from Example 1. We have $\llbracket \mu X(x).x = 0 \vee X(x - 1) \rrbracket \emptyset = \mathbf{LFP}^{(1)}(F)$, with $F = \lambda f \in D_1. \lambda n \in \mathbf{Z}. (n = 0) \sqcup f(n - 1)$. Since for any m , $F^m(\lambda x \in \mathbf{Z}. \perp) = \lambda n \in \mathbf{Z}. 0 \leq n \leq m - 1$, we have $\mathbf{LFP}^{(1)}(F) = \lambda n \in \mathbf{Z}. 0 \leq n$ (here, \leq denotes the semantic relation on integers). In contrast, $\llbracket \nu X(x).x = 0 \vee X(x - 1) \rrbracket \emptyset = \mathbf{GFP}^{(1)}(F) = \lambda n \in \mathbf{Z}. \top$. \square

2.3 Program Verification as Validity Checking of MuArith Formulas

Various verification problems for first-order recursive programs can be reduced to validity of **MuArith** formulas. We refer the reader to [22] for a general reduction schema from temporal properties to **MuArith** formulas. However, as shown in this subsection, some formulas require additional handling that motivates the need for new transformations to be presented in Section 3.

Consider the following functional program (written in the syntax of OCaml) that multiplies two numbers.

```
let rec mult(x, y) = if y=0 then 0 else x + mult(x,y-1)
```

Then, the ternary relation $Mult(x, y, r)$ that expresses “ $\text{mult}(x, y)$ terminates and returns r ” is expressed as the following **MuArith** formula:

$$\mu Mult(x, y, r). (y = 0 \wedge r = 0) \vee \exists s. (y \neq 0 \wedge r = x + s \wedge Mult(x, y - 1, s)).$$

This lets us express a partial correctness property “if $P(x, y)$ holds and $\text{mult}(x, y)$ terminates and returns r , then $Q(x, y, r)$ holds” by: $\forall x, y, r. P(x, y) \wedge \text{Mult}(x, y, r) \Rightarrow Q(x, y, r)$. It can further be rewritten to the following **MuArith** formula:

$$\forall x, y, r. \overline{P}(x, y) \vee \overline{\text{Mult}}(x, y, r) \vee Q(x, y, r), \quad (1)$$

where \overline{P} and $\overline{\text{Mult}}$ are respectively De Morgan duals of P and Mult ; $\overline{\text{Mult}}$ can be expressed by:

$$\overline{\text{Mult}}(x, y, r). (y \neq 0 \vee r \neq 0) \wedge \forall s. (y = 0 \vee r \neq x + s \vee \overline{\text{Mult}}(x, y - 1, s)).$$

The total correctness “if $P(x, y)$, then $\text{mult}(x, y)$ terminates and returns r , such that $Q(x, y, r)$ ” can be expressed by: $\forall x, y. P(x, y) \Rightarrow \exists r. \text{Mult}(x, y, r) \wedge Q(x, y, r)$, which is equivalent to the **MuArith** formula:

$$\forall x, y. \overline{P}(x, y) \vee (\exists r. \text{Mult}(x, y, r) \wedge Q(x, y, r))$$

As a special case, the termination property “if $y \geq 0$ then $\text{mult}(x, y)$ terminates” can be expressed by:

$$\forall x, y. y < 0 \vee \exists r. \text{Mult}(x, y, r). \quad (2)$$

We can also express relational properties of programs such as the equivalence of two programs. Let us consider another implementation of multiplication:

```
let mult2(x, y) =
  let rec multacc(x, y, a) = if y=0 then a else multacc(x, y-1, x+a)
  in multacc(x, y, 0)
```

Then predicate $\text{Multacc}(x, y, a, r)$ which represents “ $\text{multacc}(x, y, a)$ terminates and returns r ” can be expressed by:

$$\mu \text{Multacc}(x, y, a, r). (y = 0 \wedge r = a) \vee (y \neq 0 \wedge \text{Multacc}(x, y - 1, x + a, r)).$$

Thus, the equivalence of mult and mult2 can be expressed by: $\forall x, y, r. \text{Mult}(x, y, r) \Leftrightarrow \text{Multacc}(x, y, 0, r)$, which can be expressed by the conjunction of the **MuArith** formulas:

$$\forall x, y, r. \overline{\text{Mult}}(x, y, r) \vee \text{Multacc}(x, y, 0, r) \quad (3)$$

$$\forall x, y, r. \text{Mult}(x, y, r) \vee \overline{\text{Multacc}}(x, y, 0, r) \quad (4)$$

where $\overline{\text{Multacc}}$ is the De Morgan dual of Multacc , defined analogously to $\overline{\text{Mult}}$.

Motivation. Kobayashi et al. [22] presented a method for proving the validity of **MuArith** formulas. It can prove formula (1) valid: since there are neither μ nor \exists , it is reducible to the problem of satisfiability of CHC [4]. However, the method is not powerful enough on formulas (2) and (3) for termination and program equivalence, respectively. It first tries to eliminate existential quantifiers and μ -formulas, so that the resulting formula can be reduced to the satisfiability of

CHC. But it fails when the witness of an existential quantifier (i.e., r such that $\exists r.\varphi$) is not bounded by a linear expression, e.g., the witness for $\exists r$ is a non-linear expression $x \times y$ in the case of (2). This is unfortunate, as methods specialized on proving program termination, e.g. [18], can easily prove the termination of program `mult`. Thus, in order to exploit the advantage of the uniform approach to program verification based on `MuArith`, we need to strengthen the method for proving `MuArith` formulas.

2.4 Auxiliary Definitions

We introduce additional definitions on formulas, which will be used later in our formalization of fold/unfold-like transformations. A (k, ℓ) -context (or, just a context) is an expression obtained from an ℓ -ary predicate by replacing a k -ary predicate variable with $[]$ (in other words, a context is a predicate that may contain $[]$ as a special predicate variable). For a context C and a predicate P (that does not contain free occurrences of variables bound in C), we write $C[P]$ for the predicate obtained by replacing $[]$ with P . For example, $C \triangleq \lambda(x, y).\exists z.[](x, z, y)$ is a $(3, 2)$ -context, and $C[\lambda(x, y, z).(x > y \wedge y > z)]$ is $\lambda(x, y).\exists z.(\lambda(x, y, z).(x > y \wedge y > z))(x, z, y)$, which is equivalent to $\lambda(x, y).\exists z.x > z \wedge z > y$.

For a function $F \in \mathbf{D}_k \rightarrow \mathbf{D}_\ell$, we say that F is *continuous* if it preserves the least upper bound, i.e., $F(\bigsqcup_{f \in S} f) = \bigsqcup_{f \in S} F(f)$ for any (possibly infinite) set $S \subseteq \mathbf{D}_k$. Similarly, we say that F is *co-continuous* if it preserves the greatest lower bound, i.e., $F(\prod_{f \in S} f) = \prod_{f \in S} F(f)$. For example, $\lambda f.f \wedge g \in \mathbf{D}_0 \rightarrow \mathbf{D}_0$ and $\lambda f.f \wedge g$ is both continuous and co-continuous for any $\varphi \in \mathbf{D}_0$. In contrast, $\lambda f.\exists x.f(x) \in \mathbf{D}_1 \rightarrow \mathbf{D}_0$ is continuous but not co-continuous;⁴ $\lambda f.\forall x.f(x) \in \mathbf{D}_1 \rightarrow \mathbf{D}_0$ is co-continuous but not continuous. We say that a context C is continuous if its semantics, i.e., $\lambda f.\llbracket C[X] \rrbracket \{X \mapsto f\}$ is; analogously for co-continuity.

The following lemma (which follows immediately from the definition) provides a syntactic condition that is sufficient for the co-continuity of a context.

Lemma 1. *Let C be a (k, ℓ) -context. If C can be generated by the following syntax, then C is co-continuous.⁵*

$$C ::= [] \mid \lambda(x_1, \dots, x_k).C \mid C(a_1, \dots, a_k) \mid C \wedge \varphi \mid \varphi \wedge C \mid C \vee \varphi \mid \varphi \vee C \mid \forall x.C$$

Remark 1. The syntax and semantics of `MuArith` was defined based on hierarchical fixpoint equations (HES) in [22]. The above semantics is equivalent to that of [22], modulo the standard conversions between fixpoint formulas and HES.

⁴ In fact, let $F = \lambda f.\exists x.f(x) \in \mathbf{D}_1 \rightarrow \mathbf{D}_0$ and $S = \{\lambda x.x \geq n \mid n \in \mathbf{Z}\}$. Then $F(f) = \top$ for any $f \in S$, but $F(\prod_{f \in S} f) = F(\lambda x.\perp) = \perp$.

⁵ Here, for the sake of simplicity, we mix the syntax of contexts that yield predicates and propositions.

3 Fold/Unfold-Like Transformations

In this section, we present new fold/unfold-like transformations for **MuArith**, to enhance the power of **MuArith** validity checkers. We first informally review fold/unfold transformations for logic programming and explain what kind of transformation we wish to apply to **MuArith** formulas in Section 3.1. We then prove theorems that justify such transformations in Sections 3.2 and 3.3.

3.1 Overview of Transformations for **MuArith**

Revisiting Fold/Unfold Transformations for Logic Programming The original concept [32] is presented in the following example, where each recurrence is represented by a CHC (i.e., an implication involving uninterpreted predicates *Even* and *Odd*).

$$\begin{aligned} \text{Even}(x) &\Leftarrow x = 0 & \text{Even}(x) &\Leftarrow x > 0, \text{Even}(x - 2) \\ \text{Odd}(x) &\Leftarrow x = 1 & \text{Odd}(x) &\Leftarrow x > 0, \text{Odd}(x - 2) \end{aligned}$$

We wish to prove that $\perp \Leftarrow \text{Even}(x), \text{Odd}(x)$. Many of the existing CHC solvers, such as **HoICE** [9] and **Z3** [24], fail to prove it as they do not handle the divisibility constraints well. After defining a new predicate *EvenOdd* as $\text{EvenOdd}(x) \Leftarrow \text{Even}(x), \text{Odd}(x)$ and unfolding *Even*, we obtain the following new CHCs.

$$\text{EvenOdd}(x) \Leftarrow x = 0, \text{Odd}(x) \quad \text{EvenOdd}(x) \Leftarrow x > 0, \text{Even}(x - 2), \text{Odd}(x)$$

By unfolding *Odd*(*x*) in the first CHC, its body becomes inconsistent. By unfolding *Odd*(*x*) in the second CHC, we obtain the following new CHCs.

$$\begin{aligned} \text{EvenOdd}(x) &\Leftarrow x > 0, \text{Even}(x - 2), x = 1 \\ \text{EvenOdd}(x) &\Leftarrow x > 0, \text{Even}(x - 2), \text{Odd}(x - 2) \end{aligned}$$

By unfolding *Even*(*x* - 2), the body of the first CHC becomes inconsistent. Now, the part “*Odd*(*x* - 2), *Even*(*x* - 2)” in the second CHC matches the definition of *EvenOdd*, so we can “fold” it and obtain the following new CHC.

$$\text{EvenOdd}(x) \Leftarrow x > 0, \text{EvenOdd}(x - 2)$$

The least solution for *EvenOdd* is $\lambda x. \perp$, hence we have now obtained $\perp \Leftarrow \text{Even}(x), \text{Odd}(x)$ without synthesizing interpretations of *Even* and *Odd* over the divisibility constraints.

Transformations for **MuArith.** The above example can be reformulated in **MuArith**. Predicates *Even* and *Odd* are expressed as follows.

$$\mu \text{Even}(x). x = 0 \vee (x > 0 \wedge \text{Even}(x - 2)) \tag{5}$$

$$\mu \text{Odd}(x). x = 1 \vee (x > 0 \wedge \text{Odd}(x - 2)) \tag{6}$$

We wish to prove that $\text{Even}(x) \wedge \text{Odd}(x)$ is inconsistent, i.e. $\forall x. \overline{\text{Even}(x)} \vee \overline{\text{Odd}(x)}$ is valid where $\overline{\text{Even}}$ and $\overline{\text{Odd}}$ are:

$$\nu \overline{\text{Even}(x)}.x \neq 0 \wedge (x \leq 0 \vee \overline{\text{Even}(x-2)}) \quad (7)$$

$$\nu \overline{\text{Odd}(x)}.x \neq 1 \wedge (x \leq 0 \vee \overline{\text{Odd}(x-2)}) \quad (8)$$

Now, let $Y(x) \triangleq \overline{\text{Even}(x)} \vee \overline{\text{Odd}(x)}$, which can be rewritten as follows.

$$\begin{aligned} Y(x) &\equiv (x \neq 0 \wedge (x \leq 0 \vee \overline{\text{Even}(x-2)})) \vee (x \neq 1 \wedge (x \leq 0 \vee \overline{\text{Odd}(x-2)})) \\ &\equiv (x \leq 0 \vee x \neq 1 \vee \overline{\text{Even}(x-2)}) \wedge (x \leq 0 \vee \overline{\text{Even}(x-2)} \vee \overline{\text{Odd}(x-2)}) \\ &\equiv x \leq 0 \vee \overline{\text{Even}(x-2)} \vee \overline{\text{Odd}(x-2)} \equiv x \leq 0 \vee Y(x-2) \end{aligned}$$

Based on this, we wish to replace Y with $\nu Y(x).x \leq 0 \vee Y(x-2)$; then the validity of $\forall x.Y(x)$ would follow immediately. As we will see later in Section 3.3, this transformation is indeed sound.

Intuitively, the above transformation works as follows. Given a formula $C[X]$, which contains a fixpoint formula X defined by the equation $X = D[X]$, introduce a new predicate Y , such that $Y = C[X]$. Then, unfold X to $D[X]$ and obtain $Y = C[D[X]]$. Then, rewrite $C[D[X]]$ to a formula of the form $E[C[X]]$. By “folding” $C[X]$, we obtain $Y = E[Y]$, which serves as a new definition clause for Y . We wish to apply this kind of transformation not only to ν -only formulas like above, but also to formulas involving μ and quantifiers, as discussed below.

Recall formula (2) from Section 2.3. Let $X(x, y) \triangleq \exists r. \text{Mult}(x, y, r)$. Then,

$$\begin{aligned} X(x, y) &\equiv \exists r. ((y = 0 \wedge r = 0) \vee \exists s. (y \neq 0 \wedge r = x + s \wedge \text{Mult}(x, y - 1, s))) \\ &\equiv y = 0 \vee (y \neq 0 \wedge \exists s. \text{Mult}(x, y - 1, s)) \\ &\equiv y = 0 \vee (y \neq 0 \wedge X(x, y - 1)). \end{aligned}$$

As justified later in Section 3.2, we can then replace X with $\mu X(x, y).y = 0 \vee (y \neq 0 \wedge X(x, y - 1))$. We are then left with formula $\forall x, y. y < 0 \vee X(x, y)$, which can then be proved valid by **Mu2CHC** [22], the existing **MuArith** validity checker.

Let us also recall a generalized version of formula (3):

$$\forall x, y, a, r. \overline{\text{Mult}(x, y, r)} \vee \text{Multacc}(x, y, a, r + a),$$

which contains μ and ν . Let $Y(x, y, a, r) \triangleq \overline{\text{Mult}(x, y, r)} \vee \text{Multacc}(x, y, a, r + a)$. Then, we have:

$$\begin{aligned} Y(x, y, a, r) &\equiv ((y \neq 0 \vee r \neq 0) \wedge \forall s. (y = 0 \vee r \neq x + s \vee \overline{\text{Mult}(x, y - 1, s)})) \\ &\quad \vee (y = 0 \wedge r + a = a) \vee (y \neq 0 \wedge \text{Multacc}(x, y - 1, x + a, r + a)) \\ &\equiv (y = 0 \Rightarrow r \neq 0 \vee r + a = a) \\ &\quad \wedge (y \neq 0 \Rightarrow (\overline{\text{Mult}(x, y - 1, r - x)} \vee \text{Multacc}(x, y - 1, x + a, r + a))) \\ &\equiv y \neq 0 \Rightarrow Y(x, y - 1, x + a, r - x) \end{aligned}$$

As justified in Section 3.3, we can replace Y with $\nu Y(x, y, a, r).(y = 0 \vee Y(x, y - 1, x + a, r - x))$, giving us $\forall x, y, a, r. Y(x, y, a, r)$ immediately.

Although the above transformations are sound, the soundness of fold/unfold transformations for **MuArith** is delicate in general. For example, consider formula $\exists x.x \geq y \wedge X(x, y)$, where:

$$X \triangleq \nu X(x, y).x \geq y + 1 \wedge X(x, y + 1).$$

It is obviously false since there exists no x that satisfies $x \geq y \wedge x \geq y + 1 \wedge x \geq y + 2 \wedge \dots \equiv \forall n \geq 0.x \geq y + n$. Let $Y(y) \triangleq \exists x.x \geq y \wedge X(x, y)$. Then,

$$\begin{aligned} Y(y) &\equiv \exists x.(x \geq y \wedge x \geq y + 1 \wedge X(x, y + 1)) \\ &\equiv \exists x.(x \geq y + 1 \wedge X(x, y + 1)) \equiv Y(y + 1). \end{aligned}$$

Based on this, one may be tempted to replace Y with $\nu Y(y).Y(y + 1) \equiv \lambda y.\top$, but that is obviously wrong.

In the next two subsections, we present theorems that justify all the transformations above except the last (invalid) one.

3.2 Transformations for μ -Formulas

In this subsection, we prove a theorem that enables the replacement of a predicate of the form $C[\mu X.D[X]]$ with one of the form $\mu Y.E[Y]$ and applies it to justify the transformation for $\exists r.Mult(x, y, r)$ discussed in the previous subsection. The corresponding transformation for ν -formulas is discussed in the next subsection. The theorem is stated as follows.

Theorem 1. *Let C, D and E be (k, ℓ) , (k, k) , and (ℓ, ℓ) -contexts respectively. If $C[D[X]] \sqsupseteq_{\ell} E[C[X]]$ holds for any k -ary predicate X , then we have:*

$$C[\mu X(x_1, \dots, x_k).D[X](x_1, \dots, x_k)] \sqsupseteq_{\ell} \mu Y(y_1, \dots, y_{\ell}).E[Y](y_1, \dots, y_{\ell}).$$

The theorem follows easily from the definition of the semantics of the least fixpoint operator.

Proof. Suppose $C[D[X]] \sqsupseteq E[C[X]]$. Then, we have

$$C[\mu X(\tilde{x}).D[X](\tilde{x})] \equiv C[D[\mu X(\tilde{x}).D[X](\tilde{x})]] \sqsupseteq E[C[\mu X(\tilde{x}).D[X](\tilde{x})]].$$

Since $\mu Y(\tilde{y}).E[Y](\tilde{y})$ is the least predicate Y such that $Y \sqsupseteq E[Y]$, we have $C[\mu X(\tilde{x}).D[X](\tilde{x})] \sqsupseteq \mu Y(\tilde{y}).E[Y](\tilde{y})$ as required. \square

To see how the theorem above enables fold/unfold-like transformations, suppose that we wish to prove a formula of the form $Y \equiv C[\mu X(\tilde{x}).D[X](\tilde{x})]$. It suffices to prove $C[D[\mu X(\tilde{x}).D[X](\tilde{x})]]$, obtained by unfolding X . If the assumption $C[D[X]] \sqsupseteq E[C[X]]$ holds, we can change the goal to $E[C[\mu X(\tilde{x}).D[X](\tilde{x})]]$. Thus, by the theorem, it suffices to prove $\mu Y(\tilde{y}).E[Y](\tilde{y})$, which is obtained by “folding” $C[\mu X.D[X](\tilde{x})]$ to Y . Note that the theorem guarantees only that the transformation provides an *underapproximation* of the original predicate. A stronger condition is required for the equivalence; see Corollary 1 given later. Note also that finding an appropriate context E may not be easy in general; we discuss how to mechanically find E in Section 5.

Example 3. Recall again formula (2) from Section 2.3. Let us define C, D , and E by:

$$\begin{aligned} C &\triangleq \lambda(x, y). \exists r. [](x, y, r) \\ E &\triangleq \lambda(x, y). y = 0 \vee (y \neq 0 \wedge [](x, y - 1)) \\ D &\triangleq \lambda(x, y, r). (y = 0 \wedge r = 0) \vee \exists s. (y \neq 0 \wedge r = x + s \wedge [](x, y - 1, s)). \end{aligned}$$

Then, for any ternary predicate X , we have:

$$\begin{aligned} C[D[X]] &\equiv \lambda(x, y). \exists r. (y = 0 \wedge r = 0) \vee \exists s. (y \neq 0 \wedge r = x + s \wedge X(x, y - 1, s)) \\ &\equiv \lambda(x, y). y = 0 \vee \exists r, s. (y \neq 0 \wedge r = x + s \wedge X(x, y - 1, s)) \\ &\equiv \lambda(x, y). y = 0 \vee (y \neq 0 \wedge \exists s. X(x, y - 1, s)) \equiv E[C[X]]. \end{aligned}$$

By Theorem 1, we have $C[D[Mult]] \sqsupseteq \mu Y(x, y). y = 0 \vee (y \neq 0 \wedge Y(x, y))$. Thus, the goal $\forall x, y. y < 0 \vee \exists r. Mult(x, y, r)$ has been reduced to:

$$\forall x, y. y < 0 \vee (\mu Y(x, y). y = 0 \vee (y \neq 0 \wedge Y(x, y)))(x, y),$$

which can be proved valid by Mu2CHC. □

3.3 Fold/Unfold for ν -Formulas

We now prove a theorem that allows us to replace a predicate of the form $C[\nu X.D[X]]$ with one of the form $\nu Y.E[Y]$. It is similar to Theorem 1, but requires more conditions. Recall Lemma 1, which provides a sufficient syntactic condition for the co-continuity.

Theorem 2. *Let C, D and E be (k, ℓ) , (k, k) , and (ℓ, ℓ) -contexts respectively. Suppose that the following conditions hold: (i) $C[\top^{(k)}] \sqsupseteq_{\ell} \top^{(\ell)}$, (ii) $C[D[X]] \sqsupseteq_{\ell} E[C[X]]$, and (iii) C is co-continuous. Then $C[\nu X(x_1, \dots, x_k). D[X](x_1, \dots, x_k)] \sqsupseteq \nu Y(y_1, \dots, y_{\ell}). E[Y](y_1, \dots, y_{\ell})$.*

Proof. For $F \in \mathbf{D}_k \rightarrow \mathbf{D}_k, f \in \mathbf{D}_k$ and an ordinal γ , we define $F^{\gamma}(\top^{(k)})$ inductively by: $F^0(\top^{(k)}) = \top^{(k)}$, $F^{\gamma+1}(\top^{(k)}) = F(F^{\gamma}(\top^{(k)}))$, and $F^{\gamma}(\top^{(k)}) = \prod_{\gamma' < \gamma} F^{\gamma'}(\top^{(k)})$ if γ is a limit ordinal. By abuse of notation, we write $D^{\gamma}[\top^{(k)}]$ for $[[D]]^{\gamma}(\top^{(k)})$ if D is a (k, k) -context. Since there exists an ordinal γ such that $\nu X.D[X] = D^{\gamma}[\top^{(k)}]$ and $\nu Y.E[Y] = E^{\gamma}[\top^{(\ell)}]$, it suffices to show that $C[D^{\gamma}[\top^{(k)}]] \sqsupseteq_{\ell} E^{\gamma}[\top^{(\ell)}]$ holds for any ordinal γ , by transfinite induction on γ . The base case where $\gamma = 0$ follows immediately from the first condition. If γ is a successor ordinal $\gamma' + 1$, then

$$C[D^{\gamma}[\top]] \sqsupseteq E[C[D^{\gamma'}[\top]]] \sqsupseteq E[E^{\gamma'}[\top]] \equiv E^{\gamma}[\top].$$

Here, we have used the induction hypothesis in the second inequality. If γ is a limit ordinal, then we have:

$$C[D^{\gamma}[\top]] \equiv C[\prod_{\gamma' < \gamma} (D^{\gamma'}[\top])] \equiv \prod_{\gamma' < \gamma} C[D^{\gamma'}[\top]] \sqsupseteq \prod_{\gamma' < \gamma} E^{\gamma'}[\top] \equiv E^{\gamma}[\top].$$

Here we have used the co-continuity in the second inequality. We have thus proved $C[D^{\gamma}[\top^{(k)}]] \sqsupseteq_{\ell} E^{\gamma}[\top^{(\ell)}]$ holds for any ordinal γ . We, therefore, have $C[\nu X(\tilde{x}). D[X](\tilde{x})] \sqsupseteq \nu Y(\tilde{y}). E[Y](\tilde{y})$ as required. □

Example 4. Recall the formula $\forall x, y, a, r. \overline{Mult}(x, y, r) \vee Multacc(x, y, a, r + a)$ discussed in Section 3.1. Let us define C, D, E by:

$$\begin{aligned} C &\triangleq \lambda(x, y, a, r). [(x, y, r) \vee Multacc(x, y, a, r + a)] \\ D &\triangleq \lambda(x, y, r). ((y \neq 0 \vee r \neq 0) \wedge \forall s. (y = 0 \vee r \neq x + s \vee [(x, y - 1, s)])) \\ E &\triangleq \lambda(x, y, a, r). y = 0 \vee [(x, y - 1, x + a, r - x)] \end{aligned}$$

They satisfy all the three conditions of Theorem 2. In particular, for any ternary predicate X , we have

$$\begin{aligned} C[D[X]] &\equiv \lambda(x, y, a, r). ((y \neq 0 \vee r \neq 0) \wedge \\ &\quad \forall s. (y = 0 \vee r \neq x + s \vee X(x, y - 1, s))) \vee Multacc(x, y, a, r + a) \\ &\equiv \lambda(x, y, a, r). ((y \neq 0 \vee r \neq 0) \wedge \\ &\quad \forall s. (y = 0 \vee r \neq x + s \vee X(x, y - 1, s))) \\ &\quad \vee (y = 0 \wedge r + a = a) \vee (y \neq 0 \wedge Multacc(x, y - 1, x + a, r + a)) \\ &\equiv \lambda(x, y, a, r). y = 0 \vee X(x, y - 1, r - x) \vee \\ &\quad Multacc(x, y - 1, x + a, r + a) \\ &\equiv E[C[X]], \end{aligned}$$

based on the corresponding transformations shown in Section 3.1. We have thus $\forall x, y, a, r. \overline{Mult}(x, y, r) \vee Multacc(x, y, a, r + a) \sqsubseteq \forall x, y, a, r. (\nu Y(x, y, a, r). y = 0 \vee Y(x, y - 1, x + a, r - x))(x, y, a, r)$, and the righthand side can be proved to be valid by Mu2CHC. \square

Note that Theorems 1 and 2 guarantee the soundness of the replacement of $C[\alpha X(x_1, \dots, x_k). D[X](x_1, \dots, x_k)]$ with $\nu Y(y_1, \dots, y_\ell). E[X](y_1, \dots, y_\ell)$ (for $\alpha \in \{\mu, \nu\}$), but not completeness: the validity of $C[\alpha X(x_1, \dots, x_k). D[X](x_1, \dots, x_k)]$ does not necessarily imply that of $\nu Y(y_1, \dots, y_\ell). E[X](y_1, \dots, y_\ell)$. Actually, by combining Theorem 1 and the dual version of Theorem 2, we obtain the following corollary, which guarantees completeness under a stronger condition.

Corollary 1. *Let C, D and E be (k, ℓ) , (k, k) , and (ℓ, ℓ) -contexts respectively. Suppose that the following conditions hold: (i) $C[\perp^{(k)}] \sqsubseteq_\ell \perp^{(\ell)}$, (ii) $C[D[X]] \equiv_\ell E[C[X]]$, and (iii) C is continuous. Then $C[\mu X(x_1, \dots, x_k). D[X](x_1, \dots, x_k)] \equiv_\ell \mu Y(y_1, \dots, y_\ell). E[Y](y_1, \dots, y_\ell)$.*

4 Further Examples

In this section, we give more examples to demonstrate the utility of our transformations for relational/temporal property verification of recursive programs.

4.1 Relational Reasoning on Recursive Programs

Below we discuss an example which is beyond the reach for state-of-the-art CHC solvers (see e.g., [33], the end of Section 5).

Example 5. Consider the goal $\forall x, y, z, r. (Mult(x + y, z, r) \Rightarrow \exists s, t. Mult(x, z, s) \wedge Mult(y, z, t) \wedge r = s + t)$, which is equivalent to:

$$\forall x, y, z, r. (\overline{Mult}(x + y, z, r) \vee \exists s, t. (Mult(x, z, s) \wedge Mult(y, z, t) \wedge r = s + t)),$$

where $Mult$ and \overline{Mult} are as given in Section 2.3. The following contexts C, D , and E satisfy the following three conditions of Theorem 2.

$$\begin{aligned} C &\triangleq \lambda(x, y, z, r). [](x + y, z, r) \vee \exists s, t. (Mult(x, z, s) \wedge Mult(y, z, t) \wedge r = s + t) \\ D &\triangleq \lambda(x, z, r). (z \neq 0 \vee r \neq 0) \wedge (z = 0 \vee [](x, z - 1, r - x)) \\ E &\triangleq \lambda(x, y, z, r). (z = 0 \vee z \neq 0 \wedge [](x, y, z - 1, r - x - y)). \end{aligned}$$

By Theorem 2, we have $C[\overline{Mult}] \sqsupseteq \nu Y(x, y, z, r). E[Y](x, y, z, r) \equiv \lambda(x, y, z, r). \top$. We have thus proved that $\forall x, y, z, r. C[\overline{Mult}](x, y, z, r)$ (i.e., $\forall x, y, z, r. (Mult(x + y, z, r) \Rightarrow \exists s, t. Mult(x, z, s) \wedge Mult(y, z, t) \wedge r = s + t)$) is valid. \square

4.2 Proving Temporal Properties

Here we give an example of proving a liveness property of a recursive program by using our transformation. The example is a variation of the example discussed in [22], but it cannot be handled by their method for proving **MuArith** formulas.

Example 6. Consider the following OCaml program:

```
let rec sum n = if n=0 then 0 else n+sum(n-1)
let rec loop x = if x=0 then () else loop (x-1)
let rec repeat n = let x = sum n in loop x; repeat(n+1)
let main() = repeat 0
```

Suppose that we wish to prove that the function `repeat` is called infinitely often. The reduction from linear-time temporal property verification to **MuArith** yields the problem of determining the validity of $Repeat(0)$, where:

$$\begin{aligned} Repeat &\triangleq \nu Repeat(n). (\exists x. Sum(n, x)) \wedge (\forall x. \overline{Sum}(n, x) \vee Loop(x)) \wedge Repeat(n+1) \\ Sum &\triangleq \mu Sum(n, x). (n = 0 \wedge x = 0) \vee (n \neq 0 \wedge \exists r. Sum(n - 1, r) \wedge x = n + r) \\ Loop &\triangleq \mu Loop(x). x = 0 \vee (x \neq 0 \wedge Loop(x - 1)). \end{aligned}$$

Here, \overline{Sum} is the De Morgan dual of Sum . The validity of this formula cannot be proved by **Mu2CHC** due to the existential quantifier. Note that **Mu2CHC** replaces each existential quantifier $\exists x. \varphi$ with a bounded quantifier $\exists x \leq a. \varphi$, and a must be a linear expression. In the example above, x is not linearly bounded by n . To remove the existential quantifier, let

$$\begin{aligned} C &\triangleq \lambda n. \exists x. [](n, x) \\ E &\triangleq \lambda n. n = 0 \vee (n \neq 0 \wedge [](n - 1)) \\ D &\triangleq \lambda(n, x). (n = 0 \wedge x = 0) \vee (x \neq 0 \wedge \exists r. [](n - 1, r) \wedge x = n + r). \end{aligned}$$

Algorithm 1: Fold/unfold for disjunction

Input: Formula Φ of the form $X(f(x, y)) \vee Y(g(x, y))$, where X and Y are predicates defined by $\alpha_X, \alpha_Y \in \{\mu, \nu\}$.

Output: A formula Φ' such that $\Phi \sqsupseteq \Phi'$.

- 1 $\alpha \leftarrow$ if $\nu \in \{\alpha_X, \alpha_Y\}$ then ν else μ ;
- 2 $\bigwedge \psi_i \leftarrow \text{cnf}(\text{unfold}(\Phi))$;
- 3 **for each** ψ_i **do**
- 4 **if** ψ_i has the form $X(s_1) \vee Y(s_2) \vee \psi'_i$, $f(t_1, t_2) \equiv s_1$ and $g(t_1, t_2) \equiv s_2$ **then**
- 5 $\psi_i \leftarrow Z(t_1, t_2) \vee \psi'_i$;
- 6 **return** $\alpha Z(x, y) \cdot \bigwedge \psi_i$;

Since $C[D[X]] \sqsupseteq E[C[X]]$ holds, we can apply Theorem 1 to underapproximate $\exists x. \text{Sum}(n, x)$ by $\mu X(n).n = 0 \vee (n \neq 0 \wedge X(n-1))$. Therefore, the goal has been reduced to $\text{Repeat}'(0)$ where

$$\begin{aligned} \text{Repeat}' &\triangleq \nu \text{Repeat}'(n).X(n) \wedge (\forall x. \overline{\text{Sum}}(n, x) \vee \text{Loop}(x)) \wedge \text{Repeat}'(n+1) \\ X &\triangleq \mu X(n).n = 0 \vee (n \neq 0 \wedge X(n-1)), \end{aligned}$$

which can be proved valid by Mu2CHC automatically. \square

5 Algorithm and Evaluation

In this section, we first present an algorithm for our transformation and then outline its implementation and report on experimental results.

5.1 Algorithm

Theorems 1 and 2 given in Section 3 state sufficient conditions for our fold/unfold transformation to be sound. In this subsection, we discuss how to systematically apply the theorems and how to find a context E .

To make it easy to find E , we restrict input formulas of our transformations to those of the form $X(f(x, y)) \vee Y(g(x, y))$, $X(f(x, y)) \wedge Y(g(x, y))$, and $\exists y.X(f(x, y))$, where X and Y are predicates defined by fixpoint operators, and $f(x, y)$ and $g(x, y)$ denote (possibly sequences of) terms that may contain free variables x and y . For the sake of simplicity, we assume here that the definitions for X and Y are independent; X cannot be obtained by unfolding Y , and vice versa. Transformations for more complex formulas like the one in Example 5 can be achieved by repeatedly applying the transformations for smaller contexts.

The transformation algorithm for disjunctive formulas is shown in Algorithm 1. It takes as input a formula $\Phi = X(f(x, y)) \vee Y(g(x, y))$ and outputs an underapproximation Φ' of Φ . It can take $[(f(x, y)) \vee Y(g(x, y))]$ or $X(f(x, y)) \vee [(g(x, y))]$ as the context C and apply Theorem 2 if X or Y is

Algorithm 2: Fold/unfold for \exists

Input: Formula Φ of the form $\exists y.X(f(x,y))$, where X is a predicate defined by μ or ν .

Output: A formula Φ' such that $\Phi \sqsubseteq \Phi'$.

- 1 $\bigvee \psi_i \leftarrow \text{dnf}(\text{normalize}_{\exists}(\text{unfold}(\Phi)))$;
- 2 **for each** ψ_i **do**
- 3 **if** ψ_i has the form $(\exists z.X(s)) \wedge \psi'_i$, and $f(t_x, y) \equiv [t_z/z]s$,
 where $\mathbf{FV}(t_x) \subseteq \{x\}$, $\mathbf{FV}(t_z) \subseteq \{x, y\}$ **then**
- 4 $\psi_i \leftarrow Z(t_x) \vee \psi'_i$;
- 5 **return** $\mu Z(x). \bigvee \psi_i$;

defined by ν , and Theorem 1 otherwise (line 1). On line 2, the algorithm unfolds X and Y ⁶ and then normalizes the resulting formula to a conjunctive normal form (CNF), where quantified formulas are treated as atomic. It then applies the “fold” transformation to each conjunct ψ_i . To this end, for each ψ_i that contains $X(s_1) \vee Y(s_2)$, the algorithm finds terms t_1 and t_2 such that $X(s_1) \vee Y(s_2) \equiv X(f(t_1, t_2)) \vee Y(g(t_1, t_2))$; this is achieved by solving the unification constraints $s_1 \equiv f(x', y')$ and $s_2 \equiv g(x', y')$ modulo arithmetic theories, where x' and y' are treated as variables but x and y are treated as constants. Finally, the algorithm replaces $X(s_1) \vee Y(s_2)$ with $Z(t_1, t_2)$, where $Z(x, y)$ is a new predicate that corresponds to $X(f(x, y)) \vee Y(g(x, y))$.

We omit the transformation algorithm for conjunctive formulas since it is similar to the case above, except that the new predicate Z is bound by μ (note that condition (i) of Theorem 2 may not be satisfied), and that it converts the unfolded formula to a disjunctive normal form (DNF), instead of CNF.

The algorithm for existential formulas is shown in Algorithm 2. It unfolds X , normalizes existential quantifiers, and obtains a DNF. In the normalization of existential quantifiers, it moves existential quantifiers inwards (by using, e.g., the law $\exists x.(\psi_1 \vee \psi_2) \equiv (\exists x.\psi_1) \vee (\exists x.\psi_2)$) and eliminates them as much as possible (by using, e.g., the equality-based quantifier elimination). For each disjunct ψ_i of the form $(\exists z.X(s)) \wedge \psi'_i$, it finds t_x and t_z , such that $f(t_x, y) \equiv [t_z/z]s$ (again, by performing unification modulo arithmetic theories), and replaces the disjunct with $Z(t_x) \wedge \psi'_i$. Here, $Z(t_x)$ corresponds to $\exists y.X(f(t_x, y))$, and t_z serves as a witness for $X(f(t_x, y)) \Rightarrow \exists z.X(s)$.

5.2 Implementation and Experiments

We have implemented the transformation in a tool called `MuFolder` based on the algorithms discussed above, on top of the `AdtInd` theorem prover [37], using its routines for pattern-matching, normalization, and simplification. For the implication checks, `MUnfold` uses the `Z3` SMT solver [27]. `MuFolder` can be tested at <https://www.kb.is.s.u-tokyo.ac.jp/~koba/mu/>.

⁶ If none of ψ_i 's are changed in the loop on lines 3-5, we may backtrack and unfold X and Y more than once.

Table 1. Experiments.

#	input formula Φ	output formula Φ'
1	$\overline{Even}(x) \vee \overline{Odd}(x+1)$	$\nu Z(x).x = 0 \vee Z(x-2)$
2	$\overline{Even}(x) \vee \overline{Odd}(x)$	$\nu Z(x).(x \neq 0 \vee \overline{Even}(x-1)) \wedge Z(x-1)$
3	$\overline{Even}(x) \vee \overline{Odd}(x+1)$	$\nu Z(x).x = 0 \vee Z(x-2)$
4	$\overline{Mult}(x+y, z, r) \vee \exists s. \overline{Mult}(x, z, s)$	$\nu Z(x, y, z, r).z = 0 \vee Z(x, y, z-1, r-(x+y))$
5	$\overline{Mult}(x+y, z, r) \vee \exists s_1, s_2. \overline{Mult}(x, z, s_1) \wedge \overline{Mult}(y, z, s_2) \wedge r = s_1 + s_2$	$\nu Z(x, y, z, r).z = 0 \vee Z(x, y, z-1, r-(x+y))$
6	$\overline{Mult}(2x+3y, z, r) \vee \exists s_1, s_2. \overline{Mult}(x, z, s_1) \wedge \overline{Mult}(y, z, s_2) \wedge r = 2s_1 + 3s_2$	$\nu Z(x, y, z, r).z = 0 \vee z \neq 0 \wedge Z(x, y, z-1, r-(2x+3y))$
7	$\overline{Mult}(x, y, r) \vee \overline{Mult}(x, y, r)$	$\nu Z(x, y, r).y = 0 \vee y \neq 0 \wedge Z(x, y-1, r-x)$
8	$\overline{Mult}(x, y, r) \vee \overline{MultAcc}(x, y, a, r+a)$	$\nu Z(x, y, a, r).y = 0 \vee y \neq 0 \wedge Z(x, y-1, x+a, r-x)$
9	$\exists r. \overline{Mult}(x, y, r)$	$\mu Z(x, y).y = 0 \vee y \neq 0 \wedge Z(x, y-1)$
10	$\overline{Plus}(x+y, z, r) \vee \exists s. \overline{Plus}(x, z, s)$	$\nu Z(x, y, z, r).z = 0 \vee Z(x, y, z-1, r-1)$
11	$\overline{Plus}(4x-3y, z, r) \vee \exists s_1, s_2. \overline{Plus}(x, z, s_1) \wedge \overline{Plus}(y, z, s_2) \wedge r = 4s_1 - 3s_2$	$\nu Z(x, y, z, r).z = 0 \vee Z(x, y, z-1, r-1)$
12	$\exists r. \overline{Sum}(x, r)$	$\mu Z(x).x = 0 \vee x \neq 0 \wedge Z(x-1)$

We have evaluated **MuFolder** on several benchmarks outlined in Table 1. These benchmarks include formulas obtained from the relational and temporal verification properties; some of which have been taken from the benchmark set for Unno et al.’s induction-based CHC solver [33] and modified to include both μ and ν . We have confirmed that all the benchmark problems can be solved in our approach within a few seconds. To our knowledge, except the formulas 7, 8 (for which the method of [33] can be used) and 10,11 (for which **Mu2CHC** works), **Mu2CHC** (without our transformation) or the existing CHC solvers cannot directly prove the validity of the formulas. Note that formula 12 comes from Example 6. The combination of the transformation with **Mu2CHC** enables fully automated verification of Example 6.

6 Related Work

As already mentioned, fold/unfold transformations have been originally proposed for logic programming [32], and later extended for CHC (a.k.a. constraint logic programs) [1,17]. Those transformations have originally been proposed to speed up program execution, but recently, Mordvinov and Fedyukovich [26] and De Angelis et al. [15] shown that related transformations are also useful in the context of verification based on CHC solving. Those transformations correspond to the transformation for the ν -only fragment of **MuArith**.⁷ Our transformation can thus be considered an extension of fold/unfold-like transformations to **MuArith**, which allows alternations of least/greatest fixpoints. Sato [31] studied an extension of fold/unfold transformations for a first-order logic, where negations and quantifiers are allowed in clause bodies; thus, some mixtures of least/greatest fixpoints are allowed. The correctness of his transformation is, however, based on a three-valued logic, hence different from **MuArith**. The correctness of most of the

⁷ This is because, although the semantics of each predicate is interpreted as the least fixpoint, the predicates occur in negative positions in goal clauses.

transformations mentioned above is guaranteed by some syntactic conditions, while our transformation is based on semantic conditions.

Unno et al. [33] proposed a method for automatically solving CHC problems by using induction. Their method is based on a tailor-made proof system; hence it is difficult to integrate the method with other CHC or `MuArith` solvers (in fact, that disadvantage motivated the above-mentioned work of De Angelis et al. [15]). Their method slightly goes beyond the CHC satisfiability (or the ν -only fragment of `MuArith`) but cannot deal with complex combinations of least/greatest fixpoints and quantifiers (like $\forall x, y, z, r. (Mult(x + y, z, r) \Rightarrow \exists s, t. Mult(x, z, s) \wedge Mult(y, z, t) \wedge r = s + t)$, discussed in Section 4).

As mentioned in Section 1, fixpoint logic-based approaches to program verification (including CHC-based ones) have been drawing attention. Kobayashi et al. [22, 23, 35] have shown that temporal property verification of (higher-order) programs can be reduced to the validity checking of (higher-order) fixpoint logic formulas. They proposed a concrete method for checking validity of first-order fixpoint formulas and implemented a validity checking tool `Mu2CHC`. As discussed already, our transformations can be used to improve the capability of `Mu2CHC`. Another thread of work on a fixpoint logic-based approach to system verification is that of Parameterized Boolean Equation Systems (PBES) [21]. Actually, `MuArith` may be considered an instance of PBES, where data are restricted to integers. Groote, Willemse, and others [10, 14, 21, 30, 36] studied applications of PBES to verification of infinite state systems, and devised various techniques for solving PBES. To our knowledge, however, they have not studied fold/unfold transformations for PBES.

7 Conclusions

We have formalized fold/unfold-like transformations for a fixpoint logic, and shown that they are useful for verification of relational/temporal properties of recursive programs. We have implemented the transformations, and shown their effectiveness through experiments.

Acknowledgments. We would like to thank anonymous referees for useful comments, especially for bringing the work on PBES to our attention. This work was supported in part by the University of Tokyo-Princeton Strategic Partnership Grant, JSPS KAKENHI Grant Number JP15H05706, and NSF (USA) award FMitF 1837030.

References

1. Bensaou, N., Guessarian, I.: Transforming constraint logic programs. In: STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings. LNCS, vol. 775, pp. 33–46. Springer (1994). https://doi.org/10.1007/3-540-57785-8_129

2. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: Memory safety for systems-level code. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. LNCS, vol. 6806, pp. 178–183. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_15
3. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. LNCS, vol. 8044, pp. 869–882. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_61
4. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. LNCS, vol. 9300, pp. 24–51. Springer (2015). https://doi.org/10.1007/978-3-319-23534-9_2
5. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: 10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012. pp. 3–11. EasyChair (2012)
6. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Higher-order program verification as satisfiability modulo theories with algebraic data-types. CoRR **abs/1306.5264** (2013)
7. Bradfield, J.C.: Fixpoint alternation and the game quantifier. In: Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings. LNCS, vol. 1683, pp. 350–361. Springer (1999). https://doi.org/10.1007/3-540-48168-0_25
8. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977). <https://doi.org/10.1145/321992.321996>
9. Champion, A., Kobayashi, N., Sato, R.: Hoice: An ice-based non-linear horn clause solver. In: Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings. LNCS, vol. 11275, pp. 146–156. Springer (2018). https://doi.org/10.1007/978-3-030-02768-1_8
10. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence checking for infinite systems using parameterized boolean equation systems. In: CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings. LNCS, vol. 4703, pp. 120–135. Springer (2007). https://doi.org/10.1007/978-3-540-74407-8_9
11. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. LNCS, vol. 9206, pp. 13–29. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_2
12. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 399–410 (2011). <https://doi.org/10.1145/1926385.1926431>
13. Cook, B., Koskinen, E.: Reasoning about nondeterminism in programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 219–230. ACM (2013). <https://doi.org/10.1145/2491956.2491969>
14. Cranen, S., Luttkik, B., Willemse, T.A.C.: Proof graphs for parameterised boolean equation systems. In: CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August

- 27-30, 2013. Proceedings. LNCS, vol. 8052, pp. 470–484. Springer (2013). https://doi.org/10.1007/978-3-642-40184-8_33
15. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Solving horn clauses on inductive data types without induction. *TPLP* **18**(3-4), 452–469 (2018). <https://doi.org/10.1017/S1471068418000157>
 16. Dietsch, D., Heizmann, M., Langenfeld, V., Podelski, A.: Fairness modulo theory: A new approach to LTL software model checking. In: Proceedings of CAV 2015. LNCS, vol. 9206, pp. 49–66. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_4
 17. Etalle, S., Gabbriellini, M.: Transformations of CLP modules. *Theor. Comput. Sci.* **166**(1&2), 101–146 (1996). [https://doi.org/10.1016/0304-3975\(95\)00148-4](https://doi.org/10.1016/0304-3975(95)00148-4)
 18. Fediyukovich, G., Zhang, Y., Gupta, A.: Syntax-guided termination analysis. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018, Proceedings, Part I. LNCS, vol. 10981, pp. 124–143. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_7
 19. Gardner, P., Shepherdson, J.C.: Unfold/fold transformations of logic programs. In: Computational Logic - Essays in Honor of Alan Robinson. pp. 565–583. The MIT Press (1991)
 20. Grebenschikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416. ACM (2012). <https://doi.org/10.1145/2254064.2254112>
 21. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. *Theor. Comput. Sci.* **343**(3), 332–369 (2005). <https://doi.org/10.1016/j.tcs.2005.06.016>, <https://doi.org/10.1016/j.tcs.2005.06.016>
 22. Kobayashi, N., Nishikawa, T., Igarashi, A., Unno, H.: Temporal verification of programs via first-order fixpoint logic. In: Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings. LNCS, vol. 11822, pp. 413–436. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_20
 23. Kobayashi, N., Tsukada, T., Watanabe, K.: Higher-order program verification via HFL model checking. In: Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings. LNCS, vol. 10801, pp. 711–738. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_25
 24. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings. LNCS, vol. 8559, pp. 17–34. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_2
 25. Lubarsky, R.S.: μ -definable sets of integers. *Journal of Symbolic Logic* **58**(1), 291–313 (1993). <https://doi.org/10.2307/2275338>
 26. Mordvinov, D., Fediyukovich, G.: Synchronizing constrained horn clauses. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7–12, 2017. EPiC Series in Computing, vol. 46, pp. 338–355. EasyChair (2017)
 27. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Con-

- ference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
28. Murase, A., Terauchi, T., Kobayashi, N., Sato, R., Unno, H.: Temporal verification of higher-order functional programs. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 57–68. ACM (2016). <https://doi.org/10.1145/2837614.2837667>
 29. Nanjo, Y., Unno, H., Koskinen, E., Terauchi, T.: A fixpoint logic and dependent effects for temporal property verification. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 759–768. ACM (2018). <https://doi.org/10.1145/3209108.3209204>
 30. Orzan, S., Willemse, T.A.C.: Invariants for parameterised boolean equation systems. *Theor. Comput. Sci.* **411**(11-13), 1338–1371 (2010). <https://doi.org/10.1016/j.tcs.2009.11.001>
 31. Sato, T.: Equivalence-preserving first-order unfold/fold transformation systems. *Theor. Comput. Sci.* **105**(1), 57–84 (1992). [https://doi.org/10.1016/0304-3975\(92\)90287-P](https://doi.org/10.1016/0304-3975(92)90287-P)
 32. Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs. In: Tärnlund, S. (ed.) Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, July 2-6, 1984. pp. 127–138. Uppsala University (1984)
 33. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. LNCS, vol. 10427, pp. 571–591. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_30
 34. Urban, C., Ueltschi, S., Müller, P.: Abstract interpretation of CTL properties. In: SAS '18. LNCS, vol. 11002, pp. 402–422. Springer (2018). https://doi.org/10.1007/978-3-319-99725-4_24
 35. Watanabe, K., Tsukada, T., Oshikawa, H., Kobayashi, N.: Reduction from branching-time property verification of higher-order programs to HFL validity checking. In: Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019. pp. 22–34. ACM (2019). <https://doi.org/10.1145/3294032.3294077>
 36. Wesselink, W., Willemse, T.A.C.: Evidence extraction from parameterised boolean equation systems. In: Proceedings of the 3rd International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2018) affiliated with the International Joint Conference on Automated Reasoning (IJCAR 2018), Oxford, UK, July 18, 2018. pp. 86–100 (2018), <http://ceur-ws.org/Vol-2095/paper6.pdf>
 37. Yang, W., Fedyukovich, G., Gupta, A.: Lemma Synthesis for Automating Induction over Algebraic Data Types. In: CP 2019. LNCS, vol. 11802, pp. 600–617. Springer (2019). https://doi.org/10.1007/978-3-030-30048-7_35

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

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



Tools and Case Studies



Verifying OpenJDK's LinkedList using KeY

Hans-Dieter A. Hiep¹ , Olaf Maathuis³, Jinting Bian¹,
Frank S. de Boer¹, Marko van Eekelen², and Stijn de Gouw²



¹ CWI, Science Park 123, 1098 XG Amsterdam, The Netherlands
`{hdh,j.bian,frb}@cwi.nl`

² Open University, P.O. Box 2960, 6401 DL Heerlen, The Netherlands
`{marko.vaneekelen,stijn.degouw}@ou.nl`

³ Achmea, P.O. Box 700, 7300 HC Apeldoorn, The Netherlands
`olaf.maathuis@achmea.nl`

Abstract. As a particular case study of the formal verification of state-of-the-art, real software, we discuss the specification and verification of a corrected version of the implementation of a linked list as provided by the Java Collection framework.

Keywords: Java standard library · deductive verification · KeY · Java Modeling Language · case study · bug

1 Introduction

Software libraries are the building blocks of millions of programs, and they run on the devices of billions of users every day. Therefore, their correctness is of the utmost importance. The importance and potential of formal software verification as a means of rigorously validating state-of-the-art, real software and improving it, is convincingly illustrated by its application to TimSort, the default sorting library in many widely used programming languages, including Java and Python, and platforms like Android (see [7,9]): a crashing implementation bug was found.

The Java implementation of TimSort belongs to the Java Collection framework which provides implementations of basic data structures and is among the most widely used libraries. Nonetheless, over the years, 877 bugs in the Collections Framework have been reported in the official OpenJDK bug tracker.

Due to the intrinsic complexity of modern software, the possibility of interventions by a human verifier is indispensable for proving correctness. This holds in particular for the Java Collection library, where programs are expected to behave correctly for inputs of arbitrary size. As a particular case study, we discuss the formal verification of a corrected version of the implementation of a linked list as specified by the class `LinkedList` of the Java Collection framework in Java 8. Apart from the fact that the data structure of a linked list is one of the basic structures for storing and maintaining unbounded data, this is an interesting case study because it provides further evidence that formal verification of real software can lead to major improvements and correctness guarantees.

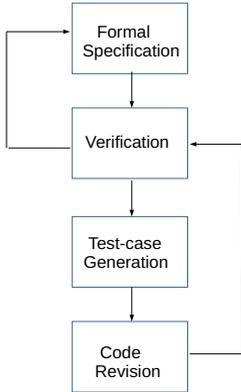


Fig. 1: Workflow

We follow the general workflow underlying the TimSort case as depicted in Fig. 1. The workflow starts with a formalisation of the informal documentation of the Java code in the Java Modeling Language [10,16]. This formalisation goes hand in hand with the formal verification: failed verification attempts can provide information about further refinements of the specs. A failed verification attempt may also indicate an error in the code, and can as such be used for the generation of test cases to detect the error at run-time.

`LinkedList` is the only `List` implementation in the Collection Framework that allows collections of unbounded size. During verification we found out that the Java linked list implementation does not correctly take into account the Java integer overflow semantics. It is exactly for large lists ($\geq 2^{31}$ items), that the implementation breaks. This basic observation gave rise to a number of test cases which show that Java’s

`LinkedList` class breaks 22 methods out of a total of 25 methods of the `List`!⁴

On the basis of these test cases we propose in Sect. 2 a code revision of the Java linked list implementation, and formally specify and verify its correctness in Sect. 3 with respect to the Java integer overflow semantics. Section 4 discusses the main challenges posed by this case study and related work.

This case study has been carried out using the state-of-the-art KeY theorem prover [3], because it formalizes the integer overflow semantics of Java and it allows to directly “load” Java programs. An archive of proof files and the KeY version used in this study is available on-line in the Zenodo repository [2].

2 `LinkedList` in OpenJDK

`LinkedList` was introduced in Java version 1.2 as part of Java’s Collection Framework in 1998. The `LinkedList` class is part of the type hierarchy of this framework: `LinkedList` implements the `List` interface, and also supports all general `Collection` methods as well as the methods from the `Queue` and `Deque` interfaces. The `List` interface provides positional access to the elements of the list, where each element is indexed by Java’s primitive `int` type.

The structure of the `LinkedList` class is shown in Listing 1. This class has three attributes: a `size` field, which stores the number of elements in the list, and two fields that store a reference to the `first` and `last` node. Internally, it uses the private static nested `Node` class to represent the items in the list. A static nested private class behaves like a top-level class, except that it is not visible outside the enclosing class (`LinkedList`, in this case). Nodes are doubly linked; each node is connected to the preceding (field `prev`) and succeeding node

⁴ We filed a bug report to Oracle’s security team. Once the report is made public by the Java maintainers, we will add the URL as metadata to our repository [2].

```

public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, ... {
    transient int size = 0;
    transient Node<E> first;
    transient Node<E> last;
    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;
        Node(Node<E> p, E i, Node<E> n) ...
    }
    ...
}

public boolean add(E e) {
    linkLast(e);
    return true;
}

void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode =
        new Node<>(l, e, null);
    last = newNode;
    if (l == null) first = newNode;
    else l.next = newNode;
    size++;
    modCount++;
}

```

Listing 1: The `LinkedList` class defines a doubly-linked list data structure.

```

public int indexOf(Object o) {
    int index = 0;
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null)
                return index;
            index++;
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
    }
    return -1;
}

```

Listing 2: The `indexOf` method searches for an element from the first node on.

(field `next`). These fields contain `null` in case no preceding or succeeding node exists. The data itself is contained in the `item` field of a node.

`LinkedList` contains 57 methods. Due to space limitations, we now focus on three characteristic methods: see Listing 1 and Listing 2. Method `add(E)` calls method `linkLast(E)`, which creates a new `Node` object to store the new item and adds the new node to the end of the list. Finally the new size is determined by unconditionally incrementing the value of the `size` field, which has type `int`. Method `indexOf(Object)` returns the position (of type `int`) of the first occurrence of the specified element in the list, or `-1` if it's not present.

Each linked list consists of a sequence of nodes. Sequences are finite, indexing of sequences starts at zero, and we write $\sigma[i]$ to mean the i th element of some sequence σ . A *chain* is a sequence σ of nodes of length $n > 0$ such that: the `prev` reference of the first node $\sigma[0]$ is `null`, the `next` reference of the last node $\sigma[n - 1]$ is `null`, the `prev` reference of node $\sigma[i]$ is node $\sigma[i - 1]$ for every index $0 < i < n$, and the `next` reference of node $\sigma[i]$ is node $\sigma[i + 1]$ for every index $0 \leq i < n - 1$. The `first` and `last` references of a linked list are either both `null` to represent the *empty* linked list, or there is some chain σ between the `first` and `last` node, viz. $\sigma[0] = \text{first}$ and $\sigma[n - 1] = \text{last}$. Figure 2 shows example instances. Also see standard literature such as Knuth's [15, Section 2.2.5].

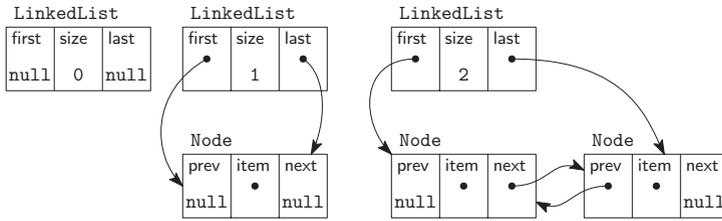


Fig. 2: Three example linked lists: empty, with a chain of one node, and with a chain of two nodes. Items themselves are not shown.

We make a distinction between the *actual* size of a linked list and its *cached* size. In principle, the size of a linked list can be computed by walking through the chain from the `first` to the `last` node, following the `next` reference, and counting the number of nodes. For performance reasons, the Java implementation also maintains a cached size. The cached size is stored in the linked list instance.

Two basic properties of doubly-linked lists are *acyclicity* and *unique first and last nodes*. Acyclicity is the statement that for any indices $0 \leq i < j < n$ the nodes $\sigma[i]$ and $\sigma[j]$ are different. First and last nodes are unique: for any index i such that $\sigma[i]$ is a node, the `next` of $\sigma[i]$ is `null` if and only if $i = n - 1$, and `prev` of $\sigma[i]$ is `null` if and only if $i = 0$. Each item is stored in a separate node, and the same item may be stored in different nodes when duplicate items are present in the list.

2.1 Integer overflow bug

The size of a linked list is encoded by a signed 32-bit integer (Java's primitive `int` type) that has a two's complement binary representation where the most significant bit is a sign bit. The values of `int` are bounded and between -2^{31} (`Integer.MIN_VALUE`) and $2^{31} - 1$ (`Integer.MAX_VALUE`), inclusive. Adding one to the maximum value, $2^{31} - 1$, results in the minimum value, -2^{31} : the carry of addition is stored in the sign bit, thereby changing the sign.

Since the linked list implementation maintains one node for each element, its size is implicitly bounded by the number of node instances that can be created. Until 2002, the JVM was limited to a 32-bit address space, imposing a limit of 4 gigabytes (GiB) of memory. In practice this is insufficient to create 2^{31} node instances. Since 2002, a 64-bit JVM is available allowing much larger amounts of addressable memory. Depending on the available memory, in principle it is now possible to create 2^{31} or more node instances. In practice such lists can be constructed today on systems with 64 gigabytes of memory, e.g., by repeatedly adding elements. However, for such large lists, at least 20 methods break, caused by signed integer overflow. For example, several methods crash with a run-time exception or exhibit unexpected behavior!

Integer overflow bugs are a common attack vector for security vulnerabilities: even if the overflow bug may seem benign, its presence may serve as a small step in a larger attack. Integer overflow bugs can be exploited more easily on large

memory machines used for 'big data' applications. Already, real-world attacks involve Java arrays with approximately $2^{32}/5$ elements [11, Section 3.2].

The `Collection` interface allows for collections with over `Integer.MAX_VALUE` elements. For example, its documentation (Javadoc) explicitly states the behavior of the `size()` method: 'Returns the number of elements in this collection. If this collection contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`'. The special case ('more than ...') for large collections is necessary because `size()` returns a value of type `int`.

When `add(E)` is called and unconditionally increments the `size` field, an overflow happens after adding 2^{31} elements, resulting in a negative `size` value. In fact, as the Javadoc of the `List` interface describes, this interface is based on integer indices of elements: 'The user can access elements by their integer index (position in the list), ...'. For elements beyond `Integer.MAX_VALUE`, it is very unclear what integer index should be used. Since there are only 2^{32} different integer values, at most 2^{32} node instances can be associated with a unique index. For larger lists, elements cannot be uniquely addressed anymore using an integer index. In essence, as we shall see in more detail below, the bounded nature of the 32-bit integer indices implies that the design of the `List` interfaces breaks down for large lists on 64-bit architectures. The above observations have many ramifications: it can be shown that 22 of 25 methods in the `List` interface are broken. Remarkably, the actual size of the linked list remains correct as the chain is still in place: most methods of the `Queue` interface still work.

2.2 Reproduction

We have run a number of test cases to show the presence of bugs caused by the integer overflow. The running Java version was Oracle's JDK8 (build 1.8.0 201-b09) that has the same `LinkedList` implementation as in OpenJDK8. Before running a test case, we set up an empty linked list instance. Below, we give an high-level overview of the test cases. Each test case uses `letSizeOverflow()` or `addElementsUntilSizeIs0()`: these repeatedly call the method `add()` to fill the linked list with `null` elements, and the latter method also adds a last element ("`this is the last element`") causing `size` to be 0 again.

1. Directly after `size` overflows, the `size()` methods returns a negative value, violating what the corresponding Javadoc stipulates: its value should remain `Integer.MAX_VALUE = 231 - 1`.

```
letSizeOverflow();
System.out.println("LinkedList.size() = " + linkedList.size() + ", actual: " + count);
// LinkedList.size() = -2147483648, actual: 2147483648
```

Clearly this behavior is in contradiction with the documentation. The actual number of elements is determined by having a field `count` (of type `long`) that is incremented each time the method `add()` is called.

2. The query method `get(int)` returns the element at the specified position in the list. It throws an `IndexOutOfBoundsException` exception when `size` is negative. From the informal specification, it is unclear what indices should be associated with elements beyond `Integer.MAX_VALUE`.

```
letSizeOverflow();
System.out.println(linkedList.get(0));
// Exception in thread "main" IndexOutOfBoundsException: Index: 0, Size: -2147483648
// at java.util.LinkedList.checkElementIndex(LinkedList.java:555) ...
```

3. The method `toArray()` returns an array containing all of the elements in this list in proper sequence (from first to last element). When `size` is negative, this method throws a `NegativeArraySizeException` exception. Furthermore, since the array size is bounded by $2^{31} - 1$ elements⁵, the contract of `toArray()` is unsatisfiable for lists larger than this. The method `Collections.sort(List<T>)` sorts the specified list into ascending order, according to the natural ordering of its elements. This method calls `toArray()`, and therefore also throws a `NegativeArraySizeException`.

```
letSizeOverflow();
Collections.sort(linkedList);
// Exception in thread "main" NegativeArraySizeException
// at java.util.LinkedList.toArray(LinkedList.java:1050)...
```

4. Method `indexOf(Object o)` returns the index of the first occurrence of the specified element in this list, or `-1` if this list does not contain the element. However due to the overflow, it is possible to have an element in the list associated to index `-1`, which breaks the contract of this method.

```
addElementUntilSizeIs0();
String last;
System.out.println("linkedList.getLast() = " + (last = linkedList.getLast()));
// linkedList.getLast() = This is the last element
System.out.println("linkedList.indexOf(" + last + ") = " + linkedList.indexOf(last));
// linkedList.indexOf(This is the last element) = -1
```

5. Method `contains(Object o)` returns true if this list contains the specified element. If an element is associated with index `-1`, it will indicate wrongly that this particular element is not present in the list.

```
addElementUntilSizeIs0();
String last;
System.out.println("linkedList.getLast() = " + (last = linkedList.getLast()));
// linkedList.getLast() = This is the last element
System.out.println("linkedList.contains(" + last + ") = " + linkedList.contains(last));
// linkedList.contains(This is the last element) = false
```

Specifically, method `letSizeOverflow()` adds 2^{31} elements that causes the overflow of `size`. Method `addElementUntilSizeIs0()` first adds $2^{32} - 1$ elements: the value of `size` is then `-1`. Then, it adds the last element, and `size` is 0 again. All elements added are `null`, except for the last element. For test cases 4 and 5, we deliberately misuse the overflow bug to associate an element with index `-1`. This means that method `indexOf(Object)` for this element returns `-1`, which according to the documentation means that the element is not present. For test cases 1, 2 and 3 we needed 65 gigabytes of memory for the JRE on a VM with 67 gigabytes of memory. For test cases 4 and 5 we needed 167 gigabytes of memory for the JRE on a VM with 172 gigabytes of memory. All test cases were carried out on a machine in a private cloud (SURFsara), which provides instances that satisfy these system requirements.

⁵ In practice, the maximum array length turns out to be $2^{31} - 5$, as some bytes are reserved for object headers, but this may vary between Java versions [11,14].

2.3 Mitigation

There are multiple directions for mitigating the overflow bug: *do not fix, fail fast, long size field* and *long or BigInteger indices*. Due to lack of space, we describe only the *fail fast* solution. This solution stays reasonably close to the original implementation of `LinkedList` and does not leave any behavior unspecified.

In the *fail fast* solution, we ensure that the overflow of `size` may never occur. Whenever elements would be added that cause the size field to overflow, the operation throws an exception and leaves the list unchanged. As the exception is triggered right before the overflow would otherwise occur, the value of `size` is guaranteed to be bounded by `Integer.MAX_VALUE`, i.e. it never becomes negative.

This solution requires a slight adaptation of the implementation: for methods that increase the `size` field, only one additional check has to be performed before a `LinkedList` instance is modified. This checks whether the result of the method causes an overflow of the `size` field. Under this condition, an `IllegalStateException` is thrown. Thus, only in states where `size` is less than `Integer.MAX_VALUE`, it is acceptable to add a single element to the list.

We shall work in a separate class called `BoundedLinkedList`: this is the improved version that does not allow more than $2^{31} - 1$ elements. Compared to the original `LinkedList`, two methods are added, `isMaxSize()` and `checkSize()`:

```
private boolean isMaxSize() {
    return size == Integer.MAX_VALUE;
}
private void checkSize() {
    if (isMaxSize())
        throw new IllegalStateException("Not enough space");
}
```

These methods implement an overflow check. The latter method is called before any modification occurs that increases the size by one: this ensures that `size` never overflows. Some methods now differ when compared to the original `LinkedList`, as they involve an invocation of the `checkSize()` method.

3 Specification and verification of BoundedLinkedList

The aim of our specification and verification effort is to verify formalizations of the given Javadoc specifications (stated in natural language) of the `LinkedList`. This includes establishing absence of overflow errors. Moreover, we restrict our attention only to the revised `BoundedLinkedList` and not to the rest of the Collection Framework or Java classes: methods that involve parameters with interface types, Java serialization or Java reflection are considered out of scope.

`(Bounded)LinkedList` inherits from `AbstractSequentialList`, but we consider its inherited methods out of scope. These methods operate on other collections such as `removeAll` or `containsAll`, and methods that have other classes as return type such as `iterator`. However, these methods call methods overridden by `(Bounded)LinkedList`, and can not cause an overflow by themselves.

We have made use of KeY's stub generator to generate dummy contracts for other classes that `BoundedLinkedList` depends on, such as for the inherited

interfaces and abstract super classes: these contracts conservatively specify that every method may arbitrarily change the heap. The stub generator moreover deals with generics by erasing the generic type parameters. For exceptions we modify their stub contract to assume that their constructors are *pure*, viz. leaving existing objects on the heap unchanged. An important stub contract is the equality method of the absolute super class `Object`, which we have adapted: we assume every object has a *side-effect free*, *terminating* and *deterministic* implementation of its equality method⁶:

```
public class Object {
    /*@ public normal_behavior
       @ requires true;
       @ ensures \result == self.equals(param0);
    @*/
    public /*@ helper strictly_pure @*/ boolean
        equals(/*@ nullable */ Object param0);
    ...
}
```

3.1 Specification

Following our workflow, we have iterated a number of times before the specifications we present here were obtained. This is a costly procedure, as revising some specifications requires redoing most verification effort. Until sufficient information is present in the specification, proving for example termination of a method is difficult or even impossible: from stuck verification attempts, and an intuitive idea of why a proof is stuck, the specification is revised.

Ghost fields. We use JML’s ghost fields: these are logical fields that for each object gets a value assigned in a heap. The value of these fields are conceptual, i.e. only used for specification and verification purposes. During run-time, this field is not present and cannot affect the course of execution. Our improved class is annotated with two ghost fields: `nodeList` and `nodeIndex`.

The type of the `nodeList` ghost field is an abstract data type of sequences, a `Key` built-in. This type has standard constructors and operations that can be used in contracts and in JML set annotations. A sequence has a length, which is finite but unbounded. The type of a sequence’s length is `\bigint`. In `Key` a sequence is untyped: all its elements are of the *any* sort, which can be any Java object reference or primitive, or built-in abstract data type. One needs to apply appropriate casts and track type information for a sequence of elements in order to cast elements of the *any* sort to any of its subsorts.

The `nodeIndex` ghost field is used as a ghost parameter with unbounded but finite integers as type. This ghost parameter is only used for specifying the behavior of the methods `unlink(Node)` and `linkBefore(Object, Node)`. The ghost parameter tracks at which index the `Node` argument is present in the `nodeList`. This information is implicit and not needed at run-time.

⁶ In reality, there are Java classes for which equality is not terminating. A nice example is `LinkedList` itself, where adding a list to itself leads to a `StackOverflowError` when testing equality with a similar instance. We consider the issue out of scope of this study as this behavior is explicitly described by the Javadoc.

Class invariant. The ghost field `nodeList` is used in the class invariant of our improved implementation, see below. We relate the fields `first` and `last` that hold a reference to a `Node` instance, and the chain between `first` and `last`, to the contents of the sequence in the ghost field `nodeList`. This allows us to express properties in terms of `nodeList`, where they reflect properties about the chain on the heap. One may compare this invariant with the description of chains as given in Sect. 2.

```

1  //@ private ghost \seq nodeList;
2  //@ private ghost \bigint nodeIndex;
3  /*@ invariant
4  @   nodeList.length == size &&
5  @   nodeList.length <= Integer.MAX_VALUE &&
6  @   (\forallall \bigint i; 0 <= i < nodeList.length;
7  @     nodeList[i] instanceof Node) &&
8  @   ((nodeList == \seq_empty && first == null && last == null)
9  @    || (nodeList != \seq_empty && first != null &&
10 @      first.prev == null && last != null &&
11 @      last.next == null && first == (Node)nodeList[0] &&
12 @      last == (Node)nodeList[nodeList.length-1])) &&
13 @   (\forallall \bigint i; 0 < i < nodeList.length;
14 @     ((Node)nodeList[i]).prev == (Node)nodeList[i-1]) &&
15 @   (\forallall \bigint i; 0 <= i < nodeList.length-1;
16 @     ((Node)nodeList[i]).next == (Node)nodeList[i+1]);
17 @*/

```

The actual size of a linked list is the length of the ghost field `nodeList`, whereas the cached size is stored in a 32-bit signed integer field `size`. On line 4, the invariant expresses that these two must be equal. Since the length of a sequence (and thus `nodeList`) is never negative, this implies that the size field never overflows. On line 5, this is made explicit: the real size of a linked list is bounded by `Integer.MAX_VALUE`. Line 5 is redundant as it follows from line 4, since a 32-bit integer never has a value larger than this maximum value. The condition on lines 6–7 requires that every node in `nodeList` is an instance of `Node` which implies it is non-null.

A linked list is either empty or non-empty. On line 8, if the linked list is empty, it is specified that `first` and `last` must be null references. On lines 9–12, if the linked list is non-empty, it is specified that `first` and `last` are non-null and moreover that the `prev` field of the first `Node` and the `next` field of the last `Node` are null. The `nodeList` must have as first element the node pointed to by `first`, and `last` as last element. In any case, but vacuously true if the linked list is empty, the `nodeList` forms a chain of nodes: lines 13–16 describe that, for every node at index $0 < i < \text{size}$, the `prev` field must point to its predecessor, and similar for successor nodes.

We note three interesting properties that are implied by the above invariant: acyclicity, unique first and unique last node. These properties can be expressed as JML formulas as follows:

```

(\forallall \bigint i; 0 <= i < nodeList.length - 1;
 (\forallall \bigint j; i < j < nodeList.length;
  nodeList[i] != nodeList[j])) &&
(\forallall \bigint i; 0 <= i < nodeList.length;
 nodeList[i].next == null <==> i = nodeList.length - 1) &&
(\forallall \bigint i; 0 <= i < nodeList.length;
 nodeList[i].prev == null <==> i = 0)

```

These properties are not literally part of our invariant, but their validity is proven interactively in KeY as a consequence of the invariant. Otherwise, we would need to reestablish also these properties each time we show the invariant holds.

Methods. All methods within scope are given a JML contract that specify its normal behavior and its exceptional behavior. As an example contract, consider the `lastIndexOf(Object)` method in Listing 3: it searches through the chain of nodes until it finds a node with an item equal to the argument. This method is interesting due to a potential overflow of the resulting index. `BoundedLinkedList` together with all method specifications are available on-line [2].

3.2 Verification

We start by giving a general strategy we apply to verify proof obligations. We also describe in more detail how to produce a single proof, in this case `lastIndexOf(Object)`. This gives a general feel how proving in KeY works. This method is neither trivial, nor very complicated to verify. In this manner, we have produced proofs for each method contract that we have specified.

Overview of verification steps. When verifying a method, we first instruct KeY to perform symbolic execution. Symbolic execution is implemented by a number of proof rules that transform modal operators on program fragments in JavaDL. During symbolic execution, the goal sequent is automatically simplified, potentially leading to branches. Since our class invariant contains a disjunction (either the list is empty or not), we do not want these cases to be split early in the symbolic execution. Thus we instruct KeY to delay unfolding the class invariant. When symbolic execution is finished, goals may still contain updated heap expressions that must be simplified further. After this has been done, one can compare the open goals to the method body and its annotations, and see whether the open goals in KeY look familiar and check whether they are true.

In the remaining part of the proof the user must find an appropriate mix between interactive and automatic steps. If a sequent is provable, there may be multiple ways to construct a closed proof tree. At (almost) every step the user has a choice between applying steps manually or automatically. It requires some experience in choosing which rules to apply manually: clever rule application decreases the size of the proof tree. Certain rules are never applied automatically, such as the cut rule. The cut rule splits a proof tree into two parts by introducing a detour, but significantly reduces the size of a proof and thus the effort required to produce it. For example, the acyclicity property can be introduced using cut.

Verification example. The method `lastIndexOf` has two contracts: one involves a `null` argument, and another involves a non-`null` argument. Both proofs are similar. Moreover, the proof for `indexOf(...)` is similar but involves the `next` reference instead of the `prev` reference. This contract is interesting, since proving its correctness shows the absence of the overflow of the `index` variable.

Proposition. `lastIndexOf(Object)` as specified in Listing 3 is correct.

Proof. Set strategy to default strategy, and set max. rules to 5,000, class axiom delayed. Finish symbolic execution on the main goal. Set strategy to 1,000 rules

```

/*@
  @ also
  @ ...
  @ public normal_behavior
  @ requires
  @   o != null;
  @ ensures
  @   \result >= -1 && \result < nodeList.length;
  @ ensures
  @   \result == -1 ==>
  @     (\forallall \bigint i; 0 <= i < nodeList.length;
  @       !o.equals(((Node)nodeList[i]).item));
  @ ensures
  @   \result >= 0 ==>
  @     (\forallall \bigint i; \result < i < nodeList.length;
  @       !o.equals(((Node)nodeList[i]).item)) &&
  @       o.equals(((Node)nodeList[\result]).item);
  @*/
public /*@ strictly_pure @*/ int
lastIndexOf(/*@ nullable @*/ Object o) {
  int index = size;
  if (o == null) {
    ...
  } else {
    /*@
      @ maintaining
      @   (\forallall \bigint i; index <= i < nodeList.length;
      @     !o.equals(((Node)nodeList[i]).item));
      @ maintaining
      @   0 <= index && index <= nodeList.length;
      @ maintaining
      @   0 < index && index <= nodeList.length ==>
      @     x == (Node)nodeList[index - 1];
      @ maintaining
      @   index == 0 <==> x == null;
      @ decreasing
      @   index;
      @ assignable
      @   \strictly_nothing;
      @*/
    for (Node x = last; x != null; x = x.prev) {
      index--;
      if (o.equals(x.item))
        return index;
    }
  }
  return -1;
}

```

Listing 3: Method `lastIndexOf(Object)` annotated with JML. Searches the list from last to first for an element. Returns `-1` if this element is not present in the list; otherwise returns the index of the node that was equal to the argument. Only the contract and branch in which the argument is non-null is shown due to space restrictions. Methods such as `indexOf`, `removeFirstOccurrence` and `removeLastOccurrence` are very similar.

and select DefOps arithmetical rules. Close all provable goals under the root. One goal remains. Perform update simplification macro on the whole sequent, perform propositional with split macro on the sequent, and close provable goals on the root of the proof tree. There is a remaining case:

- Case $index - 1 = 0 \leftrightarrow x.prev = \text{null}$: split the equivalence. First case, suppose $index - 1 = 0$, then $x = self.nodeList[0] = self.first$ and $self.first.prev = \text{null}$: solvable through unfolding the invariant and equational rewriting. Now, second case, suppose $x.prev = \text{null}$. Then, either $index = 1$ or $index > 1$ (from splitting $index \geq 1$). The first of which is trivial (close provable goal), and the second one requires instantiating quantified statements from the invariant, leading to a contradiction. Since we have supposed $x.prev = \text{null}$, but $x = self.nodeList[index - 1]$ and $self.nodeList[index - 1].prev = self.nodeList[index - 2]$ and $self.nodeList[index - 2] \neq \text{null}$. □

Interesting verification conditions. The acyclicity property is used to close verification conditions that arise as a result of potential aliasing of node instances: it is used as a separation lemma. For example, after a method that changed the `next` field of an existing node, we want to reestablish that all nodes remain reachable from the `first` through `next` fields (i.e., “connectedness”): one proves that the update of `next` only affects a single node, and does not introduce a cycle. We prove this by using the fact that two nodes instances are different if they have a different index in `nodeList`, which follows from acyclicity. Below, we sketch an argument why the acyclicity property follows from the invariant. We have a video in which we show how the argument in KeY goes, see [1, 0:55–11:30].

Proposition. *Acyclicity follows from the linked list invariant.*

Proof. By contradiction: suppose a linked list of size $n > 1$ is not acyclic. Then there are two indices, $0 \leq i < j < n$, such that the nodes at index i and j are equal. Then it must hold that for all $j \leq k < n$, the node at k is equal to the node at $k - (j - i)$. This follows from induction. Base case: if $k = j$, then node j and node $j - (j - i) = i$ are equal by assumption. Induction step: suppose node at k is equal to node at $k - (j - i)$, then if $k + 1 < n$ it also holds that node $k + 1$ equals node $k + 1 - (j - i)$: this follows from the fact that node $k + 1$ and $k + 1 - (j - i)$ are both the `next` of node $k < n - 1$ and node $k - (j - i)$. Since the latter are equal, the former must be equal too. Now, for all $j \leq k < n$, node k equals node $k - (j - i)$ in particular holds when $k = n - 1$. However, by the property that only the last node has a `null` value for `next`, and a non-last node has a non-`null` value for its `next` field, we derive a contradiction: if nodes k and $k - (j - i)$ are equal then all their fields must also have equal values, but node k has a `null` and node $k - (j - i)$ has a non-`null` next field! □

Summary of verification effort. The total effort of our case study was about 7 man months. The largest part of this effort is finding the right specification. KeY supports various ways to specify Java code: model fields/methods, pure methods, and ghost variables. For example, using pure methods, contracts are specified by expressing the content of the list before/after the method using the pure method `get(i)`, which returns the item at index i . This led to rather complex proofs: essentially it led to reasoning in terms of relational properties on programs (i.e. `get(i)` before vs `get(i)` after the method under consideration). After 2.5 man months of writing partial specifications and partial proofs in these different formalisms, we decided to go with ghost variables as this was the only formalism in which we succeeded to prove non-trivial methods.

It then took ≈ 4 man months of iterating in our workflow through (failed) partial proof attempts and refining the specs until they were sufficiently complete. In particular, changes to the class invariant were “costly”, as this typically caused proofs of all the methods to break (one must prove that all methods preserve the class invariant). The possibility to interact with the prover was crucial to pinpoint the cause of a failed verification attempt, and we used this feature of KeY extensively to find the right changes/additions to the specifications.

After the introduction of the field `nodeList`, several methods could be proved very easily, with a very low number of interactive steps or even automatically. Methods `unlink(Node)` and `linkBefore(Object, Node)` could not be proven without knowing the position of the node argument. We introduced a new ghost field, `nodeIndex`, that acts like a ghost parameter. Luckily, this did not affect the class invariant, and existing proofs that did not make use of the new ghost field were unaffected.

Once the specifications are (sufficiently) complete, we estimate that it only took approximately 1 or 1.5 man weeks to prove all methods. This can be reduced further if informal proof descriptions are given. Moreover, we have recorded a video of a 30 minute proof session where the method `unlinkLast` is proven correct with respect to its contract [1].

Proof statistics. The below table summarizes the main proof statistics for all methods. The last two columns are not metrics of the proof, but they indicate the total lines of code (LoC) and the total lines of specifications (LoSpec).

Rules	Branches	Interactive steps	Quant.ins	Contract	LoopInv	LoC	LoSpec
375,839	2,477	9,609	2,322	79	12	440	756

We found the most difficult proofs were for the method contracts of: `clear()`, `linkBefore(Object,Node)`, `unlink(Node)`, `node(int)` and `remove(Object)`. The number of interactive steps seem a rough measure for effort required. But, we note that it is not a reliable representation of the difficulty of a proof: an experienced user can produce a proof with very few interactive steps, while an inexperienced user may take many more steps. The proofs we have produced are by no means minimal.

4 Discussion

In this section we discuss some of the main challenges of verifying the real-world Java implementation of a `LinkedList`, as opposed to the analysis of an idealized mathematical linked list.

Extensive use of Java language constructs. The `LinkedList` class uses a wide range of Java language features. This includes nested classes (both static and non-static), inheritance, polymorphism, generics, exception handling, object creation and foreach loops. To load and reason about the real-world `LinkedList` source code requires an analysis tool with high coverage of the Java language, including support for the aforementioned language features.

Support for intricate Java semantics. The Java `List` interface is position based, and associates with each item in the list an index of Java's `int` type. The bugs described in Section 2.1 were triggered on large lists, in which integer overflows occurred. Thus, while an idealized mathematical integer semantics is much simpler for reasoning, it could not be used to analyze the bugs we encountered! It is therefore critical that the analysis tool faithfully supports Java's semantics, including Java's integer (overflow) behavior.

Collections have a huge state space. A Java collection is an object that contains other objects (of a reference type). Collections can typically grow to an arbitrary (but in practice, bounded) size. By their very nature, collections thus intrinsically have a large state. To make this more concrete: triggering the bugs in `LinkedList` requires at least 2^{31} elements (and 64 GiB of memory), and each element, since it is of a reference type, has at least 2^{32} values. This poses serious problems to fully automated analysis methods that explore the state space.

Interface specifications. Several of the `LinkedList` methods contain an interface type as parameter. For example, the `addAll` method takes two arguments, the second one is of the `Collection` type:

```
public boolean addAll(int index, Collection c) {
    ...
    Object[] a = c.toArray();
    ...
}
```

As KeY follows the design by contract paradigm, verification of `LinkedList`'s `addAll` method requires a contract for each of the other methods called, including the `toArray` method in the `Collection` interface. How can we specify interface methods, such as `Collection.toArray`? The stub generator generates a conservative contract: it may arbitrarily modify the heap and return *any* array. Simple conditions on parameters or the return value are easily expressed, but meaningful contracts that relates the behavior of the method to the contents of the collection require some notion of state to capture all mutations of the collection, so that previous calls to methods in the interface that contributed to the current contents of the collection are taken into account. Model fields/methods [3, Section 9.2] are a widely used mechanism for abstract specification. A model field or method is represented in a concrete class in terms of the concrete state given by its fields. In this case, as only the interface type `Collection` is known

rather than a concrete class, such a representation cannot be defined. Thus the behavior of the interface cannot be fully captured by specifications in terms of model fields/variables, including for methods such as `Collection.toArray`. Ghost variables cannot be used either, since ghost variables are updated by adding set statements in method bodies, and interfaces do not contain method bodies. This raises the question: how to specify behavior of interface methods?⁷

Verifiable code revisions. We fixed the `LinkedList` class by explicitly bounding its maximum size to `Integer.MAX_VALUE` elements, but other solutions are possible. Rather than using integers indices for elements, one could change to an index of type `long` or `BigInteger`. Such a code revision is however incompatible with the general `Collection` and `List` interfaces (whose method signatures mandate the use of integer indices), thereby breaking all existing client code that uses `LinkedList`. Clearly this is not an option in a widely used language like Java, or any language that aims to be backwards compatible.

It raises the challenge: can we find code revisions that are compatible with existing interfaces and client classes? We can take this challenge even further: can we use our workflow to find such compatible code revisions, *and are also amenable to formal verification*? The existing code in general is not designed for verification. For example, the `LinkedList` class exposes several implementation details to classes in the `java.util` package: i.e., all fields, including `size`, are package private (not private!), which means they can be assigned a new value directly (without calling any methods) by other classes in that package. This includes setting `size` to negative values. As we have seen, the class malfunctions for negative `size` values. In short, this means that the `LinkedList` itself cannot enforce its own invariants anymore: its correctness now depends on the correctness of other classes in the package. The possibility to avoid calling methods to access the `lists` field may yield a small performance gain, but it precludes a modular analysis: to assess the correctness of `LinkedList` one must now analyze all classes in the same package (!) to determine whether they make benign changes (if any) to the fields of the list. Hence, we recommend to encapsulate such implementation details, including making at least all fields `private`.

Proof reuse. Section 3.2 discussed the proof effort (in person months). It revealed that while the total effort was 6-7 person months, once the specifications are in place after many iterations of the workflow, producing the actual final proofs took only 1-2 weeks! But minor specification changes often require to redo nearly the whole proof, which causes much delay in finding the right specification. Other program verification case studies [3,4,8,9] show similarly that the main bottleneck today is specification, not verification. This calls for techniques to optimize proof reuse when the specification is slightly modified, allowing for a more rapid development of specifications.

⁷ Since the representation of classes that implement the interface is unknown in the interface itself, a particularly challenging aspect here is: how to specify the footprint of an interface method, i.e.: what part of the heap can be modified by the method in the implementing class?

Status of the challenges. Most of these challenges are still open. The challenge concerning “Interface specifications” could perhaps be addressed by defining an abstract state of an interface by using/developing some form of a trace specification that map a sequence of calls to the interface methods to a value, together with a logic to reason about such trace specifications.

The challenges related to code revisions and proof reuse are compounded for analysis tools that use very fine-grained proof representations. For example, proofs in KeY consist of actual rule applications (rather than higher level macro/strategy applications), and proof rule applications explicitly refer to the indices of the (sub) formulas the rule is applied to. This results in a fragile proof format, where small changes to the specifications or source code (such as a code refactoring) break the proof.

The KeY system covered the Java language features sufficiently to load and statically verify the `LinkedList` source code. KeY also supports various integer semantics, allowing us to analyze `LinkedList` with the actual Java integer overflow semantics. As KeY is a theorem prover (based on deductive verification), it does not explore the state space of the class under consideration, thus solving the problem of the huge state space of Java collections. We could not find any other tools that solved these challenges, so we decided at that point to use KeY.

However, other state-of-the-art systems such as Coq, Isabelle and PVS support proof *scripts*. Those proofs are described at a typically much more coarse-grained level when compared to KeY. It would be interesting to see to what extent Java language features and semantics can be handled in (extensions of) such higher level proof script languages.

4.1 Related work

Knüppel et al. [14] provide a report on the specification and verification of some methods of the classes `ArrayList`, `Arrays`, and `Math` of the OpenJDK Collections framework using KeY. Their report is mainly meant as a “stepping stone towards a case study for future research.” To the best of our knowledge, no formal specification and verification of the actual Java implementation of a linked list has been investigated. In general, the data structure of a linked list has been studied mainly in terms of pseudo code of an idealized mathematical abstraction (see [18] for an Eiffel version and [12] for a Dafny version).

This paper (and [14]) has shown that the specification and verification of actual library software poses a number of serious challenges to formal verification. In our case study, we used KeY to verify Java’s linked list. Other formalizations of Java also exists, such as Bali [17] and Jinja [13] (using the general-purpose theorem prover Isabelle/HOL), OpenJML [6] (a prover dedicated to Java programs), and VerCors [5] (focusing on concurrent Java programs, translated into Viper/Z3). However, these formalizations do not have a complete enough Java semantics to be able to analyze the bugs presented in this paper. In particular, these formalizations seem to have no built-in support for integer overflow arithmetic, although it can be added manually.

Self-references

1. Bian, J., Hiep, H.A.: Verifying OpenJDK's LinkedList using KeY: Video (2019). <https://doi.org/10.6084/m9.figshare.10033094.v2>
2. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY: Proof Files (2019). <https://doi.org/10.5281/zenodo.3517081>

References

3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification: The KeY Book, LNCS, vol. 10001. Springer (2016). <https://doi.org/10.1007/978-3-319-49812-6>
4. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification—specification is the new bottleneck. In: SSV 2012: Systems Software Verification. EPTCS, vol. 102, pp. 18–32. OPA (2012). <https://doi.org/10.4204/EPTCS.102.4>
5. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: iFM 2017: Integrated Formal Methods. LNCS, vol. 10510, pp. 102–110. Springer (2017). https://doi.org/10.1007/978-3-319-66845-1_7
6. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: F-IDE 2014: Workshop on Formal Integrated Development Environment. EPTCS, vol. 149, pp. 79–92. OPA (2014). <https://doi.org/10.4204/EPTCS.149.8>
7. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. *J. Autom. Reasoning* **62**(1), 93–126 (2019). <https://doi.org/10.1007/s10817-017-9426-4>
8. de Gouw, S., de Boer, F.S., Rot, J.: Proof Pearl: The KeY to correct and stable sorting. *J. Autom. Reasoning* **53**(2), 129–139 (2014). <https://doi.org/10.1007/s10817-013-9300-y>
9. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's `java.util.Collection.sort()` is broken: The good, the bad and the worst case. In: CAV 2015: Computer Aided Verification. LNCS, vol. 9206, pp. 273–289. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_16
10. Huisman, M., Ahrendt, W., Bruns, D., Hentschel, M.: Formal specification with JML. Tech. rep., Karlsruhe Institut für Technologie (KIT) (2014). <https://doi.org/10.5445/IR/1000041881>
11. Ieu Eauvidoum, disk noise: Twenty years of escaping the Java sandbox. Phrack Magazine (September 2018), http://www.phrack.org/papers/escaping_the_java_sandbox.html
12. Klebanov, V., Müller, P., et al.: The 1st verified software competition: Experience report. In: FM 2011: Formal Methods. LNCS, vol. 6664, pp. 154–168. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_14
13. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM TOPLAS* **28**(4), 619–695 (2006). <https://doi.org/10.1145/1146809.1146811>

14. Knüppel, A., Thüm, T., Pardyła, C., Schaefer, I.: Experience report on formally verifying parts of OpenJDK's API with KeY. In: F-IDE 2018: Formal Integrated Development Environment. EPTCS, vol. 284, pp. 53–70. OPA (2018). <https://doi.org/10.4204/EPTCS.284.5>
15. Knuth, D.E.: The art of computer programming, vol. 1. Addison-Wesley, 3rd edn. (1997) ISBN: 978-0-201-89683-4
16. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, SECS, vol. 523, pp. 175–188. Springer (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
17. Nipkow, T., von Oheimb, D.: *Javalight* is type-safe—definitely. In: POPL 1998: Principles of Programming Languages. pp. 161–170. ACM (1998). <https://doi.org/10.1145/268946.268960>
18. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: FM 2015: Formal Methods. LNCS, vol. 9109, pp. 414–434. Springer (2015). https://doi.org/10.1007/978-3-319-19249-9_26

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

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





Analysing installation scenarios of Debian packages [★]

Benedikt Becker¹ , Nicolas Jeannerod² ,
Claude Marché¹ , Yann Régis-Gianas^{2,3} ,
Mihaela Sighireanu² , and Ralf Treinen²



¹ Université Paris-Saclay, Univ. Paris-Sud, CNRS, Inria, LRI, 91405, Orsay, France

² Université de Paris, IRIF, CNRS, F-75013 Paris, France

³ Inria, F-75013 Paris, France

Abstract. The Debian distribution includes more than 28 thousand maintainer scripts, almost all of them are written in POSIX shell. These scripts are executed with root privileges at installation, update, and removal of a package, which make them critical for system maintenance. While Debian policy provides guidance for package maintainers producing the scripts, few tools exist to check the compliance of a script to it. We report on the application of a formal verification approach based on symbolic execution to find violations of some non-trivial properties required by Debian policy in maintainer scripts. We present our methodology and give an overview of our toolchain. We obtained promising results: our toolchain is effective in analysing a large set of Debian maintainer scripts and it pointed out over 150 policy violations that lead to reports (more than half already fixed) on the Debian Bug Tracking system.

Keywords: Quality Assurance · Safety Properties · Debian · Software Package Installation · Shell Scripts · High-Level View of File Hierarchies · Symbolic Execution · Feature Tree Constraints

1 Introduction

The Debian distribution is one of the oldest free software distributions, providing today 60 000 binary packages built from more than 31 000 software source packages with an official support for nine different CPU architectures. It is one of the most used GNU/Linux distributions, and serves as the basis for some derived distributions like Ubuntu.

A software package of Debian contains an archive of files to be placed on the target machine when installing the package. The package may come with a number of so-called *maintainer scripts* which are executed when installing, upgrading, or removing the package. A current version⁴ of the Debian distribution contains 28 814 maintainer scripts in 12 592 different packages, 9 771 of which

[★] This work has been partially supported by the ANR project CoLiS, contract number ANR-15-CE25-0001.

⁴ *sid* for *amd64*, including *contrib* and *non-free*, as of October 6, 2019

are completely or partially written by hand. These scripts are used for tasks like cleaning up, configuration, and repairing mistakes introduced in older versions of the distribution. Since they may have to perform any action on the target machine, the scripts are almost exclusively written in some general-purpose scripting language that allows for invoking any Unix command.

The whole installation process is orchestrated by `dpkg`, a Debian-specific tool, which executes the maintainer scripts of each package according to *scenarios*. The `dpkg` tool and the scripts require *root* privileges. For this reason, the failure of one of these scripts may lead to effects ranging from mildly annoying (like spurious warnings) to catastrophic (removal of files belonging to unrelated packages, as already reported [39]). When an execution error of a maintainer script is detected, the `dpkg` tool attempts an *error unwind*, but the success of this operation depends again on the correct behaviour of maintainer scripts. There is no general mechanism to simply undo the unwanted effects of a failed installation attempt, short of using a file system implementation providing for snapshots.

The *Debian policy* [4] aims to normalise, in natural language, important technical aspects of packages. Concerning the maintainer scripts we are interested in, it states that the standard shell interpreter is POSIX shell, with the consequence that 99% of all maintainer scripts are written in this language. The policy also sets down the control flow of the different stages of the package installation process, including attempts of error recovery, defines how `dpkg` invokes maintainer scripts, and states some requirements on the execution behaviour of scripts. One of these requirements is the *idempotency* of scripts. Most of these properties are until today checked on a very basic syntactic level (using tools like *lintian* [1]), by automated testing (like the *piuparts* suite [2]), or simply left until someone stumbles upon a bug and reports it to Debian.

The goal of our study is to improve the quality of the installation of software packages in the Debian distribution using a formal and automated approach. We focus on bug finding for three reasons. Firstly, a real Unix-like operating system is obviously too complex to be described completely and accurately by some formal model. Besides, the formal correctness properties may be difficult to apprehend by Debian maintainers especially when they are expressed on an abstract model. Finally, when a bug is detected, even on a system abstraction, one can try to reproduce it on a real system and, if confirmed, report it to the authors. This has a real and immediate impact on the quality of the software and helps to promote the usage of formal methods to a community that often is rather sceptical towards methods and tools coming from academic research.

The bugs in Debian maintainer scripts that we attempt to find may come at different levels: simple syntax errors (which may go unnoticed due to the unsafe design of the POSIX shell language), non-compliance with the requirements of the Debian policy, usage of unofficial or undocumented features, or failure of a script in a situation where it is supposed to succeed.

The challenges are multiple: The POSIX shell language is highly dynamic and recalcitrant to static analysis, both on a syntactic and semantic level. A Unix file system implementation contains many features that are difficult to

model, e.g., ownership, permissions, timestamps, symbolic links, and multiple hard links to regular files. There is an immense variety of Unix commands that may be invoked from scripts, all of which have to be modelled in order to be treated by our tools. To address properties of scripts required by the Debian policy, we need to capture the transformation done by the script on a file system hierarchy. For this, we need some kind of logic that is expressive enough, and still allows for automated reasoning methods. A particular challenge is checking the idempotency property for script execution because it requires relational reasoning. For this, we encode the semantics of a script as a logic formula specifying the relation between the input and the output of the script, and we check that it is equivalent to its composition with itself. Finally, all these challenges have to be met at the scale of tens of thousands of scripts.

The contributions of this work for this case study are:

1. A translation of Debian maintainer scripts into a language with formal semantics, and a formalisation of properties required for the execution of these scripts by the Debian policy.
2. A verification toolchain for maintainer scripts based on an existing symbolic execution engine [5,6] and a symbolic representation [26]. Some components of this toolchain have been published independently; we improve them to cope with this case study. The toolchain is free software available online [35].
3. A formal specification of the transformations done by an important set of POSIX commands [24] in *feature tree constraints* [26].
4. A number of bugs found by our method in recent versions of Debian packages.

We start in the next section with an overview of our method illustrated on a concrete example. Section 3 explains in greater detail the elements of our toolchain, the particular challenges, the hypotheses that we could make for the specific Debian use case at hand, and the solution that we have found. Section 4 presents the results we have found so far on the Debian packages, and the lessons learnt. We conclude in Section 5 by discussing additional outcomes of this study, the related and future work.

2 Overview of the case study and analysis methodology

2.1 Debian packages

Three components of a Debian binary package play an important role in the installation process: the *static content*, i.e., the archive of files to be placed on the target machine when installing the package; the lists of *dependencies* and *pre-dependencies*, which tell us which packages can be assumed present at different moments; and the *maintainer scripts*, i.e., a possibly empty subset of four scripts called `preinst`, `postinst`, `prerm`, and `postrm`. We found (Section 4.2) that 99% of the maintainer scripts in Debian are written in POSIX shell [22].

Our running example is the binary package `rancid-cgi` [31]. It comes with only two maintainer scripts: `preinst` and `postinst`. The `preinst` script is included in Fig. 1. If the symbolic link `/etc/rancid/lg.conf` exists then it is

```

1 if [ -h /etc/rancid/lg.conf ]; then
2   rm /etc/rancid/lg.conf
3 fi
4 if [ -e /etc/rancid/apache.conf ]; then
5   rm /etc/rancid/apache.conf
6 fi

```

Fig. 1. preinst script of the rancid-cgi package

removed; if the file `/etc/rancid/apache.conf` exists, no matter its type, it is also removed. Both removal operations use the POSIX command `rm` which, without options, cannot remove directories. Hence, if `/etc/rancid/apache.conf` is a directory, this script fails while trying to remove it.

We did a statistical analysis of maintainer scripts in Debian to help us design our intermediate language, see Section 4.2 for details. We found that, for instance, most variables in these scripts can be expanded statically and hence are used like constants; most `while` loops can be translated into `for` loops; recursive functions are not used at all; redirections are almost always used to discard the standard output of commands.

2.2 Managing package installation

The maintainer scripts are invoked by the `dpkg` utility when installing, removing or upgrading packages. Roughly speaking, for installation `dpkg` calls the `preinst` before the package static content is unpacked, and calls the `postinst` afterwards. For deinstallation, it calls the `prerm` before the static content is removed and calls the `postrm` afterwards. The precise sequence of script invocations and the actual script parameters are defined by informal flowcharts in the Debian policy [4, Appendix 9]. Fig. 2 shows the flowchart for the package installation. `dpkg` may be asked to: install a package that was not previously installed (Fig. 2), install a package that was previously removed but not purged, upgrade a package, remove a package, purge a package previously removed, remove and purge a package. These tasks include 39 possible execution paths, 4 of them presented in Fig. 2.

The Debian policy contains [4, Chapters 6 and 10] several requirements on maintainer scripts. This case study targets checking the requirements regarding the execution of scripts, and considers out of scope some other kinds of requirements, e.g., the permissions of script files. The requirements of interest are checked by different tools of our toolchain presented in Section 3. For example, the different ways to invoke a maintainer script are handled by the analysis of scenarios (Section 3.5) calling the scripts. Different requirements on the usage of the shell language are checked by the syntactic analysis (Section 3.1), like the usage of `-e` mode or of authorised shell features that are optional in the POSIX standard. Some of the usage requirements can be detected by a semantic analysis; this is done in our toolchain by a translation into a formally defined language, called CoLiS (Section 3.1). Finally, requirements concerning the be-

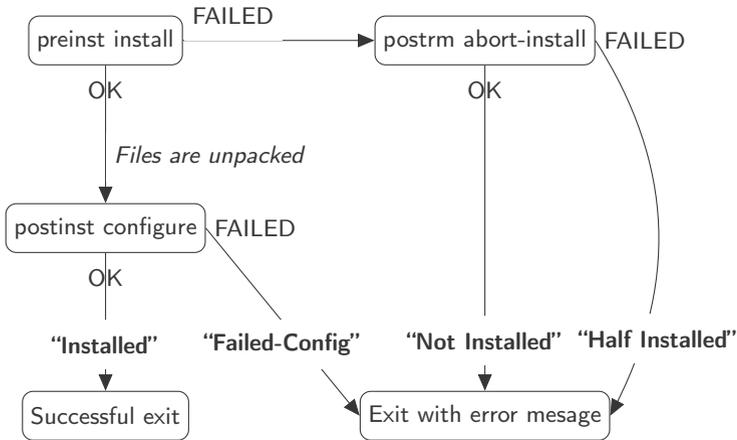


Fig. 2. Debian flowchart for installing a package [4, Appendix 9] (The states represent calls to maintainer scripts with their arguments and the status returned by `dpkg` at the end of the process is in bold.)

haviour of scripts include the usage of exit codes and the *idempotency* of scripts. The last property is difficult to formalise since it refers to possible unforeseen failures (see discussion in Section 4.4). Checking behavioural properties requires to reason about their semantics, which is done by a symbolic execution in our toolchain (Section 3.4). We also check some requirements that are simply common sense and that are not stated in the policy, e.g., invoking Unix commands with correct options. This is done by the semantic analysis (Section 3.1).

2.3 Principles and workflow of the analysis method

Our goal is to check the above properties of maintainer scripts in a formal way, by analysing each script and the composition of scripts in the execution paths exhibited by the flowcharts of `dpkg`. We call *scenario* either an execution path of `dpkg`, a single execution of a script, or a double execution of a script with the same parameters (to check idempotency); refer to Section 3.5 for more details.

The analysis should consider a variety of states for the system on which the execution takes place. Yet we assume the following hypotheses: the scripts are executed in a *root* process without concurrency with other user or *root* processes, the static content of the package is successfully unpacked, the dependencies defined by the package are present (fact checked by `dpkg`), and the `/bin/sh` command implements the standard POSIX.1-2017 Shell Command Language with the additional features described in the Debian policy [4, Chapter 10].

The components of our toolchain for the analysis of a scenario are summarised on Fig. 3 and detailed in Section 3. Given a package and one scenario, the scenario player extracts the static content and the maintainer scripts, prepares the initial *symbolic state* of the scenario, symbolically executes the steps of the scenario to

Finally, another set of generated web pages provides statistics on the coverage and the errors found for the full set of scenarios of the Debian distribution.

3 Design and implementation of the tool chain

The toolchain, as described in Fig. 3, hinges on a *symbolic execution engine* which computes the overall effect of a script on the file system as a symbolic relation between the input and the output file system. This section details this execution engine, which is composed of (i) a front-end that parses the script and translates it into a script in a formally defined intermediate language called CoLiS, and (ii) a back-end that symbolically executes the CoLiS scripts to get, for each outcome of the script, the relation between input and output file systems encoded by a *tree constraint*.

3.1 Front-end

Shell parser. The syntax of the POSIX shell language is unconventional in many aspects. For this reason, the implementation of a parser for POSIX shell cannot simply reuse the standard techniques solely based on code generators. Most of the shell implementations falls back to manually written character-level parsers, which are difficult to maintain and to trust. *morbig* [30] is a parser that tries to use code generators as much as possible to keep the parser implementation at a high level of abstraction, simplifying maintenance and improving our ability to check if it complies with the POSIX standard.

The CoLiS language. It was first presented in 2017 [23]. Its design aims to avoid some pitfalls of the shell, and to make explicit the dangerous constructions we cannot eliminate. It has a clear syntax and a *formally* defined semantics. We provide an automated and direct translation from POSIX shell. The correctness of the translation from shell to CoLiS cannot be proven formally but must be trusted based on manual review of translations and tests.

For this case study, we improved the language proposed formerly [23] to increase the number of analysed Debian maintainer scripts. First, we added a number of constructs to the language. Second, we provide a formal semantics for the new constructs and we align the previous semantics [23] to the one of the POSIX shell for a few other constructs. These changes and a complete description of the current CoLiS language are described in a technical report [6]. Fig. 5 shows the CoLiS version of the `preinst` script of the `rancid-cgi` package, shown previously in Fig. 1. Notice the syntax for string arguments and for lists of arguments that requires mandatory usage of delimiters. Generally speaking, the syntax of CoLiS is designed so as to remove potential ambiguities [6].

The toolchain for analysing CoLiS scripts is designed with formal verification in mind: the syntax, semantics, and interpreters of CoLiS are implemented using the Why3 environment [7] for formal verification. More precisely, the syntax of CoLiS is defined abstractly (as abstract syntax trees, AST for short) by an

```

1 if test [ '-h'; '/etc/rancid/lg.conf' ] then
2   rm [ '/etc/rancid/lg.conf' ]
3 fi
4 if test [ '-e'; '/etc/rancid/apache.conf' ] then
5   rm [ '/etc/rancid/apache.conf' ]
6 fi

```

Fig. 5. preinst script of the rancid-cgi package in CoLiS

algebraic datatype in Why3. Then CoLiS semantics is defined by a set of inductive predicates [6] that encodes a chiefly standard, big-step operational semantics. The semantic rules cover the contents of variables and input/output buffers used during the evaluation of a CoLiS script, but they do not specify the contents of the file system and the behaviour of POSIX commands. The judgements and rules are parameterised by bounds on the number of loop iterations and the number of (recursively) nested function calls to allow for formalising the correctness of the symbolic interpreter. The bounds are either a non-negative integer, or ∞ for unbounded execution, and keep constant throughout the evaluation of a CoLiS instruction. We refer to [6] for the details.

A concrete interpreter for the CoLiS language is implemented in Why3. Its formal specifications (preconditions and post-conditions) state the soundness of the interpreter, i.e., that any result corresponds to the formal semantics with unbounded number of loop iterations and unbounded nested function calls. The specifications are checked using automated theorem provers [23].

Translation from shell to CoLiS. This is done automatically, but it is not formally proven. Indeed, a formal semantics of shell was missing until very recently [21]. For the control flow constructs, the AST of the shell script is translated into the AST of CoLiS. For the strings (words in shell), the translation generates either a string CoLiS expression or a list of CoLiS expressions depending on the content of the shell string. This translation makes explicit the string evaluation in shell, in particular the implicit string splitting. At the present time, the translator rejects 23% of shell scripts because it does not cover the full constructs of the shell, e.g., usage of globs, variables with parameters, and advanced uses of redirections.

The conformance of the CoLiS script with the original shell script is not proven formally but tested by manual review and some automatic tests. For the latter, we developed a tool that automatically compares the results of the CoLiS interpreter on the CoLiS script with the results of the Debian default shell (`dash`) on the original shell script. This tool uses a test suite of shell scripts built to cover the whole constructs of the CoLiS language. The test suite allowed us to fix the translator and the formal semantics of CoLiS and, as an additional outcome, it revealed a lack of conformance between the Debian default shell and POSIX⁵.

⁵ <https://www.mail-archive.com/dash@vger.kernel.org/msg01683.html>

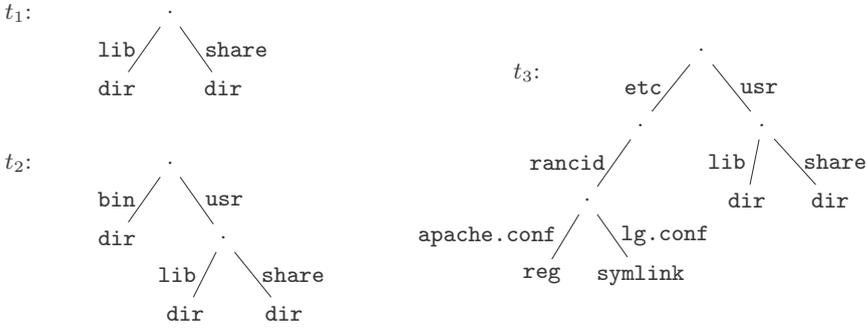


Fig. 6. Examples of feature trees showing directories (t_1), sub-directories (t_2), a regular file and a symbolic link (t_3).

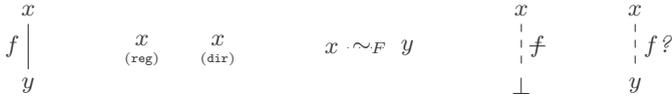


Fig. 7. Basic constraints, from left to right: a feature, a regular file node, a directory node, a tree similarity, a feature *absence*, a *maybe*

3.2 Feature trees and constraints

We employ models and logics to describe transformations of UNIX file systems. Feature trees [32,3,33] turn out to be suitable models for this case study. We have proposed a logic suitable to express file system transformations by extending previously existing logics. For the sake of space, we provide a concise overview of the model and logic used in this case study.

Feature trees. The models we consider here are trees with *features* (taken from \mathcal{F} , an infinite set of legal file names) on the edges, the *dir kind* on the nodes and any *kind* (*dir*, *reg* or *symlink*) on the leaves. Examples are given in Fig. 6.

Constraints. To specify properties of feature tree models, we modify our first order logic [26] to suit this case study’s needs. For the sake of presentation, we use a graphical representation of quantifier-free conjunctive clauses of this logic. See the technical report [24] for a detailed presentation.

The core basic constraints are presented in Fig. 7. The *feature* constraint expresses that y is a subtree of x accessible from the root of x via feature f . The *kind* constraints express that the root of a tree has the given kind (*dir*, *reg* or *symlink*). The *similarity* constraint expresses that x and y have the same children with the same names except for the children whose names are in F , a finite set of features, where they may differ.

For performance reasons, we added two more constraints; these do not increase the expressive power but help to prevent combinatorial explosion of formulas. The *absence* constraint expresses that either x is not a directory or x does not have a feature f at its root. The *maybe* constraint expresses that either x is not a directory, or it does not have a feature f at its root, or it has one that leads to y .

A model of a formula is a valuation that maps variables to feature trees. For instance, consider the valuation that associates t_1 to x , t_2 to y and t_3 to z , where t_1 , t_2 and t_3 are the trees defined in Fig. 6; it satisfies the formula in Fig. 8

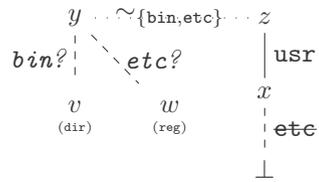


Fig. 8. A conjunctive clause

Satisfiability. We designed a set of transformation rules [26] that turns any Σ_1 -formula into an *irreducible form* that is either *false* or a satisfiable formula. This is convenient in our setting because we can detect unsatisfiable formulas as soon as possible and keep the irreducible form instead of the original formula, speeding up further computations. Our toolchain includes an implementation of this system, using an efficient representation of irreducible Σ_1 -formulas as trees themselves. Finally, the system of rules is also extended to a quantifier elimination procedure, showing that the whole first-order logic is decidable.

3.3 Specifications of UNIX commands

The specification of the UNIX commands uses our feature tree logic to express their effect on the file system. The specification formalises the description given in natural language in the POSIX standard [22, Chapter Utilities] and, for some commands, in GNU manual pages. We only specified (most of) the UNIX commands called by the maintainer scripts.

The full specification is available in a separate technical report [24]. We present here its main ingredients. A UNIX command has the form: “*cmd options paths*”, where “*cmd*” is a command name, “*options*” is a list of options, and “*paths*” is one or more absolute or relative paths (i.e., sequence of file names and symbols “.” and “..”). For each combination of command name and option, we provide a list of formulas specifying the success and failure cases. A success or failure case formula has two free variables r and r' , which represent the root of the file system before and after the command execution. For some combinations of command names and options, the specification is not provided, but computed by the symbolic execution of a CoLiS script. This script captures the command behaviour by calling other (primitive) commands.

Path resolution. An important ingredient in command specification is the constraint encoding the resolution of a path in the file system. For this, we define a predicate `resolve(r, cwd, p, z)` stating that “when the root of the file system is r and the current working directory is the sequence of features cwd , the path p resolves and goes to variable z ”. The constraint defining this predicate is a Σ_1

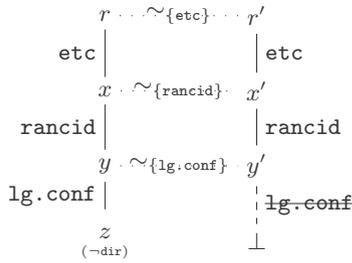


Fig. 9. Specification of success case for `rm /etc/rancid/lg.conf`

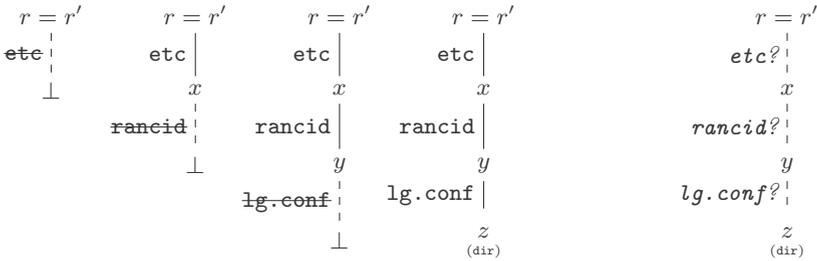


Fig. 10. Specification of error cases of `rm /etc/rancid/lg.conf`: explicit cases on the left, compact specification on the right

conjunction of basic constraints; it does not deal with symbolic link files on the path. For example, the constraint `resolve(r, cwd, /etc/rancid/lg.conf, z)` is represented by the path starting from r and ending in z in Fig. 9.

For some commands, a failure of path resolution may cause the failure of the command. To specify these failure cases, we have to use the negation of the predicate `resolve`, which generates a number of clauses which is linear in the length of the resolved path. Fig. 10 shows, in the three left-most constraints, the error cases for the resolution of the path to `/etc/rancid/lg.conf`. Because the internal representation of formulas keeps only conjunctive clauses, this may produce a state explosion of constraints when the command uses several paths. To obtain a compact internal representation of these error cases, we employ the *maybe* shorthand, as shown on the right of Fig. 10.

Let us consider the command `rm /etc/rancid/lg.conf`. Its specification includes one success case, given on Fig. 9: the resolution of the path `/etc/rancid/lg.conf` succeeded in the initial file system denoted by r , and the resulting file system, denoted by r' is similar to r except for the absence of the feature `lg.conf`. The specification also includes one error case given on Fig. 10, where the path cannot be resolved to a regular path, and therefore the initial and final file systems are the same.

It is important to notice that specifications of commands are *parameterised* by their path(s) argument(s): for each concrete value of such paths, an appropri-

ate constraint is produced. This fact is essential for using our symbolic engine, because the variables of a constraint denote nodes of the file system, but there is no notion of variable denoting file names or paths.

3.4 Analysis by symbolic execution

With a similar approach as for the concrete interpreter (Section 3.1), we designed and implemented a symbolic interpreter for the CoLiS language in Why3. Guided by a proof-of-concept symbolic interpreter for a simple IMP language [5], the main design choices for the symbolic interpreter for CoLiS are:

- Variables are *not* interpreted abstractly: when executing an installation script, the concrete values of the variables are known. On the other hand, the state of the file system is not known precisely, and it is represented symbolically using a feature tree constraint.
- The symbolic engine is generic with respect to the utilities: their specifications in terms of symbolic input/output relations are taken as parameters.
- The number of loop iterations and the number of (recursively) nested function calls [6] is bounded a priori, the bound is given by a global parameter set at the interpreter call.

The Why3 code for the symbolic interpreter is annotated with post-conditions to express that it computes an *over-approximation* [5] of the concrete states that are reachable without exceeding the given bound on loop iterations. This property is formally proven using automated provers. The OCaml code is automatically extracted from Why3, and provides an executable symbolic interpreter with strong guarantees of soundness with respect to the concrete formal semantics.

Notice that our symbolic engine neither supports parallel executions, nor file permissions or file timestamps. This is another source of over-approximation, but also under-approximation, meaning that our approach can miss bugs whose triggering relies on the former features.

The symbolic interpreter provides a symbolic semantics for the given script: given an initial symbolic state that represents the possible initial shape of the file system, it returns a triple of sets of symbolic input/output relations, respectively for normal result, error result (corresponding to non-zero exit code) and result when a loop limit is reached. Error results are unexpected for Debian maintainer scripts, and these cases have to be inspected manually. To help this inspection, a visualisation of symbolic relations was designed, as already described in Fig. 4.

3.5 Scenarios

So far, we have presented how we analyse individual maintainer scripts. In reality, the Debian policy specifies in natural language in which order and with which arguments these scripts are invoked during package installation, upgrade, or removal (see, for instance, Fig. 2). We have specified these scenarios in a loop-free custom language. These scenarios define what happens after the success or

the failure of a script execution. They also specify when the static content is unpacked. Furthermore, our toolchain allows to define the assumptions that can be made on an initial filesystem before executing a scenario, for instance the File System Hierarchy Standard [38]. Our toolchain reports on packages that may remain in an unexpected state after the execution of one of these scenarios.

For instance, the installation scenario of the package *rancid-cgi* may leave that package in the state *not-installed*, which is reported by our toolchain using the diagram in Fig. 4.

4 Results and impact

4.1 Coverage of the case study

The tools used and the datasets analysed during the current study are available in the Zenodo repository [36].

We execute the analysis on a machine equipped with 40 hyperthreaded Intel Xeon CPU @ 2.20GHz, and 750GB of RAM. To obtain a reasonable execution time, we limit the processing of one script to 60 seconds and 8GB of RAM. The time limit might seem low, but the experience shows that the few scripts (in 30 packages) that exceed this limit actually require hours of processing because they make a heavy use of `dpkg-maintscript-helper`. On our corpus of 12 592 packages with 28 814 scripts, the analysis runs in about half an hour.

All of those scripts that are syntactically correct with respect to the POSIX standard (99.9%) are parsed successfully by our parser. The translation of the parsed scripts into our intermediary language CoLiS succeeds for 77% of them; the translation fails mainly because of the use of globs, variables with parameters and advanced uses of redirections.

Our toolchain then attempts to run 113 328 scenarios (12 592 packages with scripts, 9 scenarios per package). Out of those, 45 456 scenarios (40%) are run completely and 13 149 (12%) partially. This is because scenarios have several branches and although a branch might encounter failure, we try to get some information on execution of other branches. For the same reason, one scenario might encounter several failures. In total, we encounter 67 873 failures. The origins of failures are multiple, but the two main ones are (i) trying to execute a scenario that includes a script that we cannot convert (28% of failures), or (ii) the scripts might use commands unsupported by our tools, or unsupported features of supported commands (71% of failures).

Among the scenarios that we manage to execute at least partially, 19 reach an unexpected end state. These are potential bugs. We have examined them manually to remove false positives due to approximations done by our methodology or the toolchain. We discuss in Section 4.3 the main classes of true bugs revealed by this process.

4.2 Corpus mining

The latest version of the Debian *sid* distribution on which we ran our tools dates from October 6, 2019. It contains 60 000 packages, 12 592 of which contain at

Table 1. Bugs found between 2016 and 2019 in Debian *sid* distributions

<i>Bugs</i>	<i>Closed</i>	<i>Detected by</i>	<i>Reports</i>	<i>Examples</i>
95	56	parser	[9]	not using <code>-e</code> mode
6	4	parser & manual	[15]	unsafe or non-POSIX constructs
34	24	corpus mining	[8,10]	wrong options, mixed redirections
9	7	translation	[11]	wrong test expressions
5	2	symbolic execution	[13,17,15]	try to remove a directory with <code>rm</code>
3	3	formalisation	[12]	bug in <code>dpkg-maintscript-helper</code>
152	96			

least one maintainer script, which leads to 28 814 scripts. In total, these scripts contain 442 364 source lines of code, 15 lines on average, and up to 1 138 for the largest script. Among them we find 220 `bash` scripts, 2 `dash` scripts, 14 `perl` scripts, and one ELF executable – the rest are POSIX shell scripts.

In the process of designing our tools, and in order to validate our hypotheses, we ran statistical analysis on this corpus of scripts. The construction of our tool for statistical analysis is described in a technical report [25] where we also detail a few of our findings. To summarise, analysing the corpus revealed that:

- Most variables in scripts were used as constants: only 3 008 scripts contain variables whose value actually changes.
- There are no recursive functions in the whole corpus.
- There are 2 300 scripts that include a while loop. 93% of the while loops occur in a pipe reading the output of `dpkg -L` and are an idiosyncrasy that is proper to some shell languages. They can be translated to “foreach” loops in a properly typed language.
- The huge majority of redirections are used to hide the standard output or merge it into the error output.

This analysis had an important impact on the project by guiding the design choices of CoLiS, which Unix commands we should specify and in which order, etc. This also helped us to discover a few bugs, e.g., scripts invoking Unix commands with invalid options.

4.3 Bugs found

We ran our toolchain on several snapshots of the Debian *sid* distribution taken between 2016 and 2019, the latest one being October 6, 2019. We reported over this period a total of 152 bugs to the Debian Bug Tracking System [37]. Some of them have immediately been confirmed by the package maintainer (for instance, [16]), and 96 of them have already been resolved.

Table 1 summarises the main categories of bugs we reported. Simple lexical analysis already detects 95 violations of the Debian Policy, for instance scripts that do not specify the interpreter to be used, or that do not use the `-e` mode [9]. The shell parser (Section 3.1) detects 3 scripts that use shell constructs not allowed by the POSIX standard, or in a context where the POSIX standard states

that the behaviour is undefined [15]. There are also 3 miscellaneous bugs, like using unsafe shell constructs. The mining tool (Section 4.2) detects 5 scripts that invoke Unix commands with wrong options and 29 scripts that mix up redirection of standard-output and standard-error. The translation from the shell to the CoLiS language (Section 3.1) detects 9 scripts with wrong test expressions [11]. These may stay unnoticed during superficial testing since the shell confuses, when evaluating the condition of an if-then-else, an error exception with the Boolean value *False*. Inspection of the symbolic semantics extracted by the symbolic execution (Section 3.4) finds 5 scripts with semantic errors. Among these is the bug [16] of the package *rancid-cgi* already explained in Section 2.4. During the formalisation of Debian tools (see Section 3.3), we found 3 bugs. These include in particular a bug [12] in the `dpkg-maintscript-helper` command which is used 10 306 times in our corpus of maintainer scripts, and was fixed in the meantime.

4.4 Lessons learnt

One basic problem when trying to analyse maintainer scripts is to understand precisely the meaning of the policy document. For instance, one of the more intriguing requirements is that maintainer scripts have to be idempotent (Section 6.2 in [4]). While it is common knowledge that a mathematical function f is idempotent when $f(f(x)) = f(x)$ for any x , the meaning is much less clear in the context of Debian maintainer scripts as the policy goes on to explain “If the first call failed, or aborted half way through for some reason, the second call should merely do the things that were left undone the first time, if any, and exit with a success status if everything is OK.” We suppose that this refers to causes of error external to the script itself (power failure, full disk, etc.), and that there might be an intervention by the system administrator between the two invocations. Since we cannot even explain in natural language what precisely that means, let alone formalise it, we decided to model at the moment only a rough under-approximation of that property that only compares executions by their exit code. This allowed us to detect a bug [14].

We found that identifying bugs in maintainer scripts always requires human examination. Automated tools allow to point out potential problems in a large corpus, but deciding whether such a problem actually deserves a bug report, and of what severity level, requires some experience with the Debian processes. This is most visible with semantic bugs in scripts, since an error exit code does not imply that there is a bug. Indeed, if a script detects a situation it cannot handle then it *must* signal an error and produce a useful error message. Deciding whether a detected error case is justified or accidental requires human judgement.

Filing bug reports demands some caution, and observance of rules and common practices in the community. For instance, the Debian Developers Reference [18] requires approval by the community before so-called *mass bug filing*. Consequently, we always sought for advice before sending batches of bugs, either on the Debian developers mailing list, or during Debian conferences.

5 Conclusion

The corpus of Debian maintainer scripts is an interesting case study for analysis due to its size, the challenging features of the scripting language, and the relational properties it requires to analyse. The results are very promising. First, we reported 152 bugs [37] to the Debian Bug Tracking system, 96 of which have already been resolved by Debian maintainers. Second, the toolchain performs the analysis of a package in seconds and of the full distribution in less than an hour, which makes it fit for integration in the workflow of Debian maintainers or for quality assurance at the level of the whole distribution. Integration of our toolchain in the `lintian` tool will not be possible since it would add a lot of external dependencies to that tool, and since the reports generated by our tool still require human evaluation (see Section 4.4).

This study had several additional outcomes. The toolchain includes tools for parsing and light static analysis of shell scripts [30], an engine for the symbolic execution of imperative languages based on first-order logics representation of program configurations [5], and an efficient decision procedure for feature tree logics. We also provide a formal specification of POSIX commands used in Debian scripts in terms of a first-order logic [24].

We are not aware of a project dealing with this kind of problem or obtaining comparable results. To our knowledge, the only existing attempt to analyse a complete corpus of package maintainer scripts was done in the context of the Mancoosi project [19]. In this work, the analysis, mainly syntactic, resulted in a set of building blocks used in maintainer scripts that may be used in a DSL. In a series of papers [20,28,29], Ntzik et al. consider the formal reasoning on the POSIX scripts manipulating the file system based on (concurrent) separation logic. Not only do they employ a different logic (a second-order logic), but they also focus on (manual) proof techniques for correctness and not on automatic techniques for finding bugs. Moreover, they consider general scripts and properties that are not relational (like idempotency). There have been few attempts to formalise the shell. Greenberg [21] recently offers an executable formal semantics of POSIX shell that will serve as a foundation for shell analysis tools. Abash [27] contains a formalisation of parts of the bash language and an abstract interpretation tool for the analysis of arguments passed by scripts to Unix commands; this work focused on identifying security vulnerabilities.

The successful outcome of this case study revealed new challenges that we aim to address in future work. In order to increase the coverage of our analysis and the acceptance by Debian maintainers, the translation from shell should cover more features, additional Unix commands should be formally specified, and the model should capture more features of the file system, e.g., permissions, or symbolic links. The efficiency of the analysis can still be improved by using a more compact representation of disjunctive constraints in feature tree logics or by exploiting the genericity of the symbolic execution engine to include other logic based symbolic representations that may be more efficient and precise. Finally, we want to use the computed constraints on scenarios to check new properties of scripts like equivalence of behaviours.

References

1. Lintian. <https://lintian.debian.org>
2. Piuparts. <https://piuparts.debian.org/>
3. Aït-Kaci, H., Podelski, A., Smolka, G.: A feature-based constraint system for logic programming with entailment. *Theor. Comput. Sci.* **122**(1–2), 263–283 (Jan 1994)
4. Allbery, R., Whitton, S.: Debian policy manual (Oct 2019), <https://www.debian.org/doc/debian-policy/>
5. Becker, B., Marché, C.: Ghost Code in Action: Automated Verification of a Symbolic Interpreter. In: Chakraborty, S., A.Navas, J. (eds.) *Verified Software: Tools, Techniques and Experiments*. Lecture Notes in Computer Science (2019), <https://hal.inria.fr/hal-02276257>
6. Becker, B., Marché, C., Jeannerod, N., Treinen, R.: Revision 2 of CoLiS language: formal syntax, semantics, concrete and symbolic interpreters. Technical report, HAL Archives Ouvertes (Oct 2019), <https://hal.inria.fr/hal-02321743>
7. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)* **17**(6), 709–727 (2015). <https://doi.org/10.1007/s10009-014-0314-5>, <http://hal.inria.fr/hal-00967132/en>, see also <http://toccata.lri.fr/gallery/fm2012comp.en.html>
8. Debian Bug Tracker: dibbler-server: postinst contains invalid command. Debian Bug Reports 841934 (Oct 2016), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=841934>
9. Debian Bug Tracker: authbind: maintainer script(s) not using strict mode. Debian Bug Report 866249 (Jun 2017), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=866249>
10. Debian Bug Tracker: dict-freedict-all: postinst script has a wrong redirection. Debian Bug Report 908189 (Sep 2018), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=908189>
11. Debian Bug Tracker: python3-neutron-fwaas-dashboard: incorrect test in postrm. Debian Bug Report 900493 (May 2018), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=900493>
12. Debian Bug Tracker: [dpkg-maintscript-helper] bug in finish_dir_to_symlink. Debian Bug Report 922799 (Feb 2019), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=922799>
13. Debian Bug Tracker: ndiswrapper: when "postrm purge" fails it may have deleted some config files. Debian Bug Report 942392 (Oct 2019), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942392>
14. Debian Bug Tracker: oz: non-idempotent postrm script. Debian Bug Report 942395 (Oct 2019), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942395>
15. Debian Bug Tracker: preinst script not posix compliant. Debian Bug Report 925006 (Mar 2019), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=925006>
16. Debian Bug Tracker: rancid-cgi: preinst may fail and not rollback a change. Debian Bug Report 942388 (Oct 2019), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=942388>
17. Debian Bug Tracker: sgml-base: preinst may fail *silently*. Debian Bug Report 929706 (May 2019), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=929706>
18. Developer's Reference Team: Debian developers reference (Oct 2019), <https://www.debian.org/doc/manuals/developers-reference/>

19. Di Cosmo, R., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Supporting software evolution in component-based FOSS systems. *Science of Computer Programming* **76**(12), 1144–1160 (2011). <https://doi.org/10.1016/j.scico.2010.11.001>
20. Gardner, P., Ntzik, G., Wright, A.: Local reasoning for the POSIX file system. In: *European Symposium On Programming. Lecture Notes in Computer Science*, vol. 8410, pp. 169–188. Springer (2014). https://doi.org/10.1007/978-3-642-54833-8_10
21. Greenberg, M., Blatt, A.J.: Executable formal semantics for the POSIX shell. *CoRR* **abs/1907.05308** (2019), <http://arxiv.org/abs/1907.05308>
22. IEEE, The Open Group: The open group base specifications issue 7. <http://pubs.opengroup.org/onlinepubs/9699919799/> (2018)
23. Jeannerod, N., Marché, C., Treinen, R.: A Formally Verified Interpreter for a Shell-like Programming Language. In: *9th Working Conference on Verified Software: Theories, Tools, and Experiments. Lecture Notes in Computer Science*, vol. 10712 (2017), <https://hal.archives-ouvertes.fr/hal-01534747>
24. Jeannerod, N., Régis-Gianas, Y., Marché, C., Sighireanu, M., Treinen, R.: Specification of UNIX utilities. Technical report, HAL Archives Ouvertes (Oct 2019), <https://hal.inria.fr/hal-02321691>
25. Jeannerod, N., Régis-Gianas, Y., Treinen, R.: Having fun with 31.521 shell scripts. Tech. rep., HAL Archives Ouvertes (2017), <https://hal.archives-ouvertes.fr/hal-01513750>
26. Jeannerod, N., Treinen, R.: Deciding the first-order theory of an algebra of feature trees with updates. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) *9th International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science*, vol. 10900, pp. 439–454. Springer, Oxford, UK (Jul 2018), <https://hal.archives-ouvertes.fr/hal-01807474>
27. Mazurak, K., Zdancewic, S.: ABASH: finding bugs in bash scripts. In: *Workshop on Programming Languages and Analysis for Security*. pp. 105–114 (2007)
28. Ntzik, G., Gardner, P.: Reasoning about the POSIX file system: local update and global pathnames. In: *Object-Oriented Programming, Systems, Languages and Applications*. pp. 201–220. ACM (2015). <https://doi.org/10.1145/2814270.2814306>
29. Ntzik, G., da Rocha Pinto, P., Sutherland, J., Gardner, P.: A concurrent specification of POSIX file systems. In: *European Conference on Object-Oriented Programming. LIPIcs*, vol. 109, pp. 4:1–4:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018). <https://doi.org/10.4230/LIPIcs.ECOOP.2018.4>
30. Régis-Gianas, Y., Jeannerod, N., Treinen, R.: Morbig: A static parser for POSIX shell. In: Pearce, D., Mayerhofer, T., Steimann, F. (eds.) *ACM SIGPLAN International Conference on Software Language Engineering*. pp. 29–41. Boston, MA, USA (Nov 2018). <https://doi.org/10.1145/3276604.3276615>, <https://hal.archives-ouvertes.fr/hal-01890044>
31. Rosenfeld, R.: Package rancid-cgi: looking glass cgi based on rancid tools (2019), <https://packages.debian.org/en/sid/rancid-cgi>
32. Smolka, G.: Feature constraint logics for unification grammars. *Journal of Logic Programming* **12**, 51–87 (1992)
33. Smolka, G., Treinen, R.: Records for logic programming. *Journal of Logic Programming* **18**(3), 229–258 (Apr 1994)
34. The CoLiS project: The CoLiS bench. <http://ginette.informatique.univ-paris-diderot.fr/~niols/colis-batch/>
35. The CoLiS project: The CoLiS toolchain. <https://github.com/colis-anr>

36. The CoLiS project: Artifact for Analysing installation scenarios of Debian Packages. Zenodo Repository (Feb 2020). <https://doi.org/10.5281/zenodo.3678390>
37. The Debian Project: Bugs tagged colis, <https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=colis-shparser;users=treinen@debian.org>
38. The Linux Foundation: Filesystem hierarchy standard, version 3.0 (Mar 2015), <https://refspecs.linuxfoundation.org>
39. Ucko, A.M.: cmigrep: broken emacsen-install script. Debian Bug Report 431131 (Jun 2007), <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=431131>

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

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





Endicheck: Dynamic Analysis for Detecting Endianness Bugs

Roman Kápl and Pavel Parížek

Department of Distributed and Dependable Systems,
Faculty of Mathematics and Physics, Charles University,
Prague, Czech Republic



Abstract. Computers store numbers in two mutually incompatible ways: little-endian or big-endian. They differ in the order of bytes within representation of numbers. This ordering is called endianness. When two computer systems, programs or devices communicate, they must agree on which endianness to use, in order to avoid misinterpretation of numeric data values.

We present Endicheck, a dynamic analysis tool for detecting endianness bugs, which is based on the popular Valgrind framework. It helps developers to find those code locations in their program where they forgot to swap bytes properly. Endicheck requires less source code annotations than existing tools, such as Sparse used by Linux kernel developers, and it can also detect potential bugs that would only manifest if the given program was run on computer with an opposite endianness. Our approach has been evaluated and validated on the Radeon SI Linux OpenGL driver, which is known to contain endianness-related bugs, and on several open-source programs. Results of experiments show that Endicheck can successfully identify many endianness-related bugs and provide useful diagnostic messages together with the source code locations of respective bugs.

1 Introduction

Modern computers represent and store numbers in two mutually incompatible ways: little-endian (with the least-significant byte first) or big endian (the most-significant byte first). The byte order is also referred to as *endianness*.

Processor architectures typically define a *native endianness*, in which the processor stores all data. When two computer systems or programs exchange data (e.g., via a network), they must first agree on which endianness to use, in order to avoid misinterpretation of numeric data values. Also devices connected to computers may have control interfaces with endianness different from the host's native endianness.

Therefore, programs communicating with other computers and devices need to swap the bytes inside all numerical values to the correct endianness. We use the term *target endianness* to identify the endianness a program should use for data exchanged with a particular external entity. Note that in some cases it is not necessary to know whether the target endianness is actually little-endian or big-endian. When the knowledge is important within the given context, we use the term *concrete endianness*.

If the developer forgets to transform data into the correct target endianness, the bug can often go unnoticed for a long time because software is nowadays usually developed and tested on the little-endian x86 or ARM processor architecture. For example,

© The Author(s) 2020

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12079, pp. 254–270, 2020.

https://doi.org/10.1007/978-3-030-45237-7_15

if two identical programs running on a little-endian architecture communicate over the network using a big-endian protocol, a missing byte-order transformation in the same place in code will not be observed. Our work on this project was, in the first place, motivated by the following concrete manifestation of the general issue described in the previous sentence. The Linux OpenGL driver for Radeon SI graphics cards (the Mesa 17.4 version) does not work on big-endian computers due to an endianness-related bug¹, as the first author discovered when he was working on an industrial project that involved PowerPC computers in which Radeon graphic cards should be deployed.

We are aware of few approaches to detection of endianness bugs, which are based on static analysis and manually written source code annotations. An example is Sparse [11], a static analysis tool used by Linux kernel developers to identify code locations where byte-swaps are missing. The analysis performed by Sparse works basically in the same way as type checking for C programs, and relies on the usage of specialized *bitwise* data types, such as `_le16` and `_be32`, for all variables with non-native endianness. Integers with different concrete endianness are considered by Sparse as having mutually incompatible types, and the specialized types are also not compatible with regular C integer types. In addition, macros like `_le32_to_cpu` are provided to enable safe conversion between values of the bitwise integer types and integer values of regular types. Such macros are specially annotated so that the analysis can recognize them, and developers are expected to use only those macros.

The biggest advantage of bitwise types is that a developer cannot assign a regular native endianness integer value to a variable of a bitwise type, or vice versa. Their nature also prevents the developer from using them in arithmetic operations, which do not work correctly on values with non-native byte order. On the other hand, a significant limitation of Sparse is that developers have to properly define the bitwise types for all data where endianness matters, and in particular to enable identification of data with concrete endianness — Sparse would produce imprecise results otherwise. Substantial manual effort is therefore required to create all the bitwise types and annotations.

Our goals in this whole project were to explore an approach based on dynamic analysis, and to reduce the amount of necessary annotations in the source code of a subject program. We present Endicheck, a dynamic analysis tool for detecting endianness bugs that is implemented as a plugin for the Valgrind framework [6]. The main purpose of the dynamic analysis performed by Endicheck is to track endianness of all data values in the running subject program and report when any data leaving the program has the wrong endianness. The primary target domain consists of programs written in C or C++, and in which developers need to explicitly deal with endianness of data values.

While the method for endianness tracking that we present is to a large degree inspired by dynamic *taint* analyses (see, e.g., [8]), our initial experiments showed that usage of existing taint analysis techniques and tools does not give good results especially with respect to precision. For example, an important limitation of the basic taint analysis, when used for endianness checking, is that it would report false positives on data that needs no byte-swapping, such as single byte-sized values. Therefore, we had to modify and extend the existing taint analysis algorithms for the purpose of endianness checking. During our work on Endicheck, we also had to solve many associated tech-

¹ https://bugs.freedesktop.org/show_bug.cgi?id=99859

nical challenges, especially regarding storage and propagation of metadata that contain the endianness information — this includes, for example, precise tracking of single-byte values.

Endicheck is meant to be used only during the development and testing phases of the software lifecycle, mainly because it incurs a substantial runtime overhead that is not adequate for production deployment. Before our Endicheck tool can be used, the subject program needs to be modified, but only to inform the analysis engine where the byte-order is being swapped and where data values are leaving the program. In C and C++ programs, byte-order swapping is typically done by macros provided in the system C library, such as `htons/htonl` or those defined in the `endian.h` header file. Thus only these macros need to be annotated. During the development of Endicheck, we redefined each of those macros such that the custom variant calls the original macro and defines necessary annotations — for examples, see Figure 1 in Section 4 and the customized header file `inet.h`². Similarly, data also tend to leave the program only through few procedures. For some programs, the appropriate place to check for correct endianness is the `send/write` family of system calls.

Endicheck is released under the GPL license. Its source code is available at <https://github.com/rkapl/indicheck>.

The rest of the paper is structured as follows. Section 2 begins with a more thorough overview of the dynamic analysis used by Endicheck, and then it provides details about the way endianness information for data values are stored and propagated — this represents our main technical contribution, together with evaluation of Endicheck on the Radeon SI driver and several other real programs that is described in Section 5. Besides that, we also provide some details about the implementation of Endicheck (Section 3) together with a short user guide (Section 4).

2 Dynamic Analysis for Checking Endianness

We have already mentioned that the dynamic analysis used by Endicheck to detect endianness bugs is a special variant of taint analysis, since it uses and adapts some related concepts. In the rest of this paper, we use the term *endianness analysis*.

2.1 Algorithm Overview

Here we present a high-level overview of the key aspects of the endianness analysis. Like common taint and data-flow analysis techniques (see, e.g., [4] and [8]), our dynamic endianness analysis tracks flow of data through program execution, together with some metadata attached to specific data values. The analysis needs to attach metadata to all memory locations for which endianness matters, and maintain them properly. Metadata associated with a sequence of bytes (memory locations) that makes a numeric data value then capture its endianness. Similarly to many dynamic analyses, the metadata are stored using a mechanism called *shadow memory* [7] [9]. We give more details about the shadow memory in Section 2.2.

² <https://github.com/rkapl/indicheck/blob/master/indicheck/ec-overlay/arpa/inet.h>

Although we mostly focus on checking that the program being analyzed does not transmit data of incorrect endianness to other parties, there is also the opposite problem: ensuring that the program does not use data of other than native endianness. For this reason, our endianness analysis could be also used to check whether all operands of an arithmetic instruction have the correct native endianness — this is important because arithmetic operations are unlikely to produce correct results otherwise. Note, however, that checking of native endianness for operands has not yet been implemented in the Endicheck tool.

The basic principle behind the dynamic endianness analysis is to watch instructions as they are being executed and check endianness at specific code locations, such as the calls of I/O functions. We use the term *I/O functions* to identify all system calls and other functions that encapsulate data exchange between a program and external entities (e.g., writing or reading data to/from a hard disk, or network communication) in a specific endianness. When the program execution reaches the call of an I/O function, Endicheck checks whether all its arguments have the proper endianness. Note that the user of Endicheck specifies the set of I/O functions by annotations (listed in Section 4).

In order to properly maintain the endianness information stored in the shadow memory, our analysis needs to track almost every instruction being executed during the run of a subject program. The analysis receives notifications about relevant events from the Valgrind dynamic analysis engine. All the necessary code for tracking individual instructions (processing the corresponding events), updating endianness metadata (inside the shadow memory), and checking endianness at the call sites of I/O functions, is added to the subject program through dynamic binary instrumentation. Further technical details about the integration of Endicheck into Valgrind are provided later in Section 3.

Two distinguishing aspects of the endianness analysis — the format of metadata stored in the shadow memory and the way metadata are propagated during the analysis of program execution — are described in the following subsections.

2.2 Shadow Memory

A very important requirement on the organization and structure of shadow memory was full transparency for any C/C++ or machine code program. The original layout of heap and stack has to be preserved during the analysis run, since Endicheck (and Valgrind in general) targets C and C++ programs that typically rely on the precise layout of data structures in memory. Consequently, Endicheck cannot allocate the space for shadow memory (metadata) within the data structures of the analyzed program.

When designing the endianness analysis, we decided to use the mechanism supported by Valgrind [7], which allows client analyses to store a tag for each byte in the virtual address space of the analyzed program without changing its memory layout. This mechanism keeps a translation table (similar to page tables used by operating systems) that maps memory pages to *shadow pages* where the metadata are stored.

The naive approach would be to follow the same principles as taint analyses, i.e. reuse the idea of taint bits, and mark each byte of memory as being either of native endianness or target endianness. However, our endianness analysis actually uses a richer format of metadata and individual tags, which improves the analysis precision.

Rich Metadata Format. In this format of metadata, each byte of memory and each processor register is annotated with one of the following tags that represent available knowledge about the endianness of stored data values.

- **native:** The default endianness produced, for example, by arithmetic operations.
- **target:** Used for data produced by annotated byte-swapping function.
- **byte-sized:** Marks the first byte of a multi-byte value (e.g., an integer or float).
- **unknown:** Endianness of uninitialized data (e.g., newly allocated memory blocks).

In addition to these four tags, each byte of memory can also be annotated with the **empty** flag, indicating that the byte's value is zero. Now we give more details about the meaning of these tags, and discuss some of the associated challenges.

Single-byte values. Our approach to precise handling of single-byte values is motivated by the way arithmetic operations are processed. Determining the correct size of the result of an arithmetic operation (in terms of the number of actually used bytes) is difficult in practice, because compilers often choose to use instructions that operate on wider types than actually specified by the developer in program source code. This means the analysis cannot, in some cases, precisely determine whether the result of an arithmetic operation has only a single byte. Our solution is to always mark the least-significant byte of the result with the tag **byte-sized**. Such an approach guarantees that if only the least-significant byte of an integer value is actually used, it does not trigger any endianness errors when checked, because the respective memory location is not tagged as **native**. On the other hand, if the whole integer value is really used (or at least more than just the least-significant byte), there is one byte marked with the tag **byte-sized** and the rest of the bytes are marked as **native**, thus causing an endianness error when checked.

Empty byte flag. Usage of the empty flag helps to improve performance of the endianness analysis when processing byte-shuffling instructions, because all operations with empty flags are simpler than operations with the actual values. However, this flag can be soundly used only when the operands are byte-wise disjoint, i.e. when each byte is zero (empty) in at least one of the operands. Arithmetic operations are handled in a simplified way — they never mark bytes as empty in the result. Consequently, while the empty tag implies that the given byte is zero, the reverse implication does not hold.

Unknown tag. We introduced the tag **unknown** in order to better handle data values, for which the analysis cannot say whether they are already byte-swapped. Endicheck uses this tag especially for uninitialized data. Values marked with the tag **unknown** are not reported as erroneous by default, but this behavior is configurable. We discuss other related problems, concerning especially precision, below in Section 2.4.

2.3 Propagation of Metadata

An important aspect of the endianness analysis is that data values produced by the subject program are marked as having the native endianness by default. This behav-

ior matches the prevailing case, because data produced by most instructions (e.g., by arithmetic operations) and constant values can be assumed to have native endianness.

In general, metadata are propagated upon execution of an instruction according to the following policy:

- Arithmetic operations always produce native-endianness result values.
- Data manipulation operations (e.g., load and store) propagate tags from their operands to results without any changes.

Endicheck correctly passes metadata also through routines such as `memcpy` and certain byte-shuffling operations (e.g., `shift <<=` and `>>=`). Complete details for all categories of instructions and routines are provided in the master thesis of the first author [3].

The only way to create data with the target tag is via explicit annotation from the user. Specifically, the user needs to add annotations to *byte-swapping* functions in order to set the target tag on return values.

2.4 Discussion: Analysis Design and Precision

The basic scenario that is obviously supported by our analysis is the detection of endianness bugs when the target and native endianness are different. However, the design of our analysis ensures that it can be useful even in cases when the native endianness is the same as the target endianness. Although byte-swapping functions then become identities, the endianness analysis can still find data that would not be byte-swapped if the endiannities were different — it can do this by setting the respective tags when data pass through the byte-swapping functions. In addition, the endianness analysis can be also used to detect the opposite direction of errors — programs using non-native endianness data values (e.g., received as input) without byte-swapping them first.

Endicheck does not handle constants and immediate values in instructions very well, since the analysis cannot automatically recognize their endianness and therefore cannot determine whether the data need byte-swapping or not. Constants stored in the data section of a binary executable represent the main practical problem to the analysis, because the data section does not have any structure — it is just a stream of bytes. Our solution is to mark data sections initially with the tag unknown. If this is not sufficient, a user must annotate the constants in the program source code to indicate whether they already have the correct endianness.

A possible source of false bug reports are unused bytes within a block of memory that has undefined content, unless the memory was cleared with 0s right after its allocation. This may occur, for example, when some fields inside C structures have specific alignment requirements. Some space between individual fields inside the structure layout is then unused, and marked either with the tag unknown or with the tag left over from the previous content of the memory block.

3 Implementation

We distribute the Endicheck tool in the form of an open source software package that was initially created as a fork of the Valgrind source code repository. Although tools

and plugins for Valgrind can be maintained as separate projects, forking allowed us to make changes to the Valgrind core and use its build/test infrastructure. Within the whole source tree of Endicheck, which includes the forked Valgrind codebase, the code specific to Endicheck is located in the `endicheck` directory. It consists of these modules:

- `ec_main`: tool initialization, command-line handling and routines for translation to/from intermediate representation;
- `ec_errors`: error reporting, formatting and deduplication;
- `ec_shadow`: management of the shadow memory, storing of the endianness metadata, protection status and origin tracking information (see below);
- `ec_util`: utility functions for general use and for manipulation with the metadata;
- `endicheck.h`: public API with annotations to be used in programs by developers.

In the rest of this section, we briefly describe how Endicheck uses the Valgrind infrastructure and a few other important features. Additional technical details about the implementation are provided in the master thesis of the first author [3].

Usage of Valgrind infrastructure. Endicheck depends on the Valgrind core (i) for dynamic just-in-time instrumentation [6] of a target binary program and (ii) for the actual dynamic analysis of program execution. The subject binary program is instrumented with code that carries out all the tasks required by our endianness analysis — especially recording of important events and tracking information about the endianness of data values. When implementing the Endicheck plugin, we only had to provide code doing the instrumentation itself and define what code has to be injected at certain locations in the subject program. Note also that for the analysis to work correctly and provide accurate results, Valgrind instruments all components of the subject program that may possibly handle byte-swapped data, including application code, the system C library and other libraries. During the analysis run, Valgrind notifies the Endicheck plugin about execution of relevant instructions and Endicheck updates the information about endianness of affected data values accordingly. Besides instrumentation and the actual dynamic analysis, other features and mechanisms of the Valgrind framework used by Endicheck include: utility functions, origin tracking, and developer-friendly error reporting.

Origin tracking [1] is a mechanism that can help users in debugging the endianness issues. An error report contains two stack traces: one identifies the source code location of the call to the I/O function where the wrong endianness of some data value was detected, and the second trace, provided by origin tracking, identifies the source code location where the value has originated. In Endicheck, the origin information (identifier of the stack trace and execution context) is stored alongside the other metadata in the shadow memory for all values. We decided to use this approach because almost all values need origin tracking, since they can be sources of errors — in contrast to Memcheck, where only the uninitialized values can be sources of errors.

During our experiments with the Radeon SI OpenGL driver (described in Section 5.1), we have noticed that the driver maps the device memory into the user-space process. In that case, there is no single obvious point where to check the endianness of data that leave the program through the mapped memory. To solve this problem and support memory-mapped I/O, we extended our analysis to automatically check endianness at all writes to regions of the mapped device memory. We implemented this feature

in such a way that each byte of a device memory region is tagged with a special flag protected — then, Endicheck can find very quickly whether some region of memory is mapped to a device or not. Note that the flag is associated with a memory location, while the endianness tags (described in Section 2.2) are associated with *data values*. Therefore, the special flag is not copied, e.g. when execution of `memcpy` is analyzed; it can be only set explicitly by the user.

4 User Guide

The recommended way to install Endicheck is building from the source code. Instructions are provided in the README file at the project web site. When Endicheck has been installed, a user can run it by executing the following command:

```
valgrind --tool=endicheck [OPTIONS...] PROGRAM ARGS...
```

Origin tracking is enabled by the option `-track-origins=yes`.

Annotations In order to analyze a given program, some annotations typically must be added into the program source code. A user of Endicheck has to mark the byte-swapping functions and the I/O functions (through which data values are leaving the program), because these functions cannot be reliably detected in an automated way.

The specific annotations are defined in the C header file `endicheck.h`. Here follows the list of supported annotations, together with explanation of their meaning:

- `EC_MARK_ENDIANITY(start, size, endianness)`
This annotation marks a region of memory from `start` to `start+size-1` as having the given endianness. It should be used in byte-swapping functions. Target endianness is represented by the symbol `EC_TARGET`.
- `EC_CHECK_ENDIANITY(start, size, msg)`
This annotation enforces a check that a memory region from `start` to `start+size-1` contains only data with *any* or *target* endianness. It should be used in I/O functions. *Unknown* endianness is allowed by passing the `-allow-unknown` option.
- `EC_PROTECT_REGION(start, size)`
Marks the given region of memory as protected. This should be used for mapped regions of device memory.
- `EC_UNPROTECT_REGION(start, size)`
Marks the given memory region as unprotected.
- `EC_DUMP_MEM(start, size)`
Dumps endianness of a memory region. This is useful for debugging.

Figure 1 shows an example program that demonstrates usage of the most important annotations (`EC_MARK` and `EC_CHECK`). If the call to `htobe32` inside `main` is removed, Endicheck will report an endianness bug. This example also demonstrates possible ways to easily annotate standard functions, like `htobe32` and `write`.

```

#include <valgrind/endicheck.h>

uint32_t htobe32(uint32_t x) {
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    x = bswap_32(x);
#endif
    EC_MARK_ENDIANITY(&x, sizeof(x), EC_TARGET);
    return x;
}

int ec_write(int file, const void *buf, size_t count) {
    EC_CHECK_ENDIANITY(buf, count, NULL);
    return write(file, buf, count);
}
#define write ec_write

int main() {
    uint32_t x = 0xDEADBEEF;
    x = htobe32(x);
    write(0, &x, sizeof(x));
    return 0;
}

```

Fig. 1. Small example program with Endicheck annotations.

5 Evaluation

We evaluated the Endicheck tool — namely its ability to find endianness bugs, precision and overhead — by the means of a case study on the Radeon SI driver, several open-source programs and a standardized performance benchmark. For the Radeon SI driver and each of the open-source programs, we provide a link to its source code repository (and identification of the specific version that we used for our evaluation) within the artifact that is referenced from the project web site.

5.1 Case Study

Our case study is Radeon SI, the Linux OpenGL driver for Radeon graphics cards, starting with the SI (Southern Islands) line of cards and continuing to the current models.

Since these Radeon cards are little-endian, the driver must byte-swap all data when running on a big-endian architecture such as PowerPC. However, the Radeon SI driver (in the Mesa 17.4 version) does not perform the necessary byte-swapping operations, and therefore simply does not work in the case of PowerPC — it crashes either the GPU or OpenGL programs using the driver. In particular, endianness bugs in this version of the Radeon SI driver cause the Glxgears demo on PowerPC to crash. We give more details about the bugs we have found in Section 5.2.

An important feature of the whole Linux OpenGL stack is that all layers, including the user-space program, communicate not only using calls of library functions and

system calls, but they also extensively use mapping of the device memory directly into the user process. Given such an environment, Endicheck has to correctly handle (1) the flow of data through the whole OpenGL stack by instrumenting all the libraries used, and (2) communication through the shared memory that is used by the driver. This is why the support for mapped memory in Endicheck, through marking of device memory with a special flag, as described above in Section 3, is essential.

5.2 Search for Bugs

For the purpose of evaluating Endicheck’s ability to find endianness bugs, we picked a diverse set of open-source programs (in addition to the Radeon SI driver), including the following: BusyBox, OpenTTD, X.Org and ImageMagick. All programs are listed in Table 1. The only criterion was to select programs written in C that communicate over the network or store data in binary files, since only such programs may possibly contain endianness bugs. We also document our experience with fixing the endianness bugs in the Radeon SI driver and other programs.

One of the stated goals for Endicheck was to reduce the number of annotations that a user must add into the program source code in order to enable search for endianness bugs. Therefore, below we report the relevant measurements and discuss whether (and to what degree) this goal has been achieved.

In the rest of this section, first we discuss application of Endicheck on the Radeon SI driver (our case study) and then we present results for other programs.

Radeon SI case study. Within our case study, we have used the Glxgears demo program as a test harness for the Radeon SI driver. Initially we have run Glxgears on the x86 architecture, and after fixing all the issues found and reported by Endicheck, we moved the same graphics card to a PowerPC host computer and continued testing there.

In the case of the Radeon SI driver, all byte-swapping functions are located in a single file of one library (Gallium) on the OpenGL stack. Therefore, to enable search for endianness bugs in Radeon SI, we had to make just two changes: (1) annotate the function `radeon_drm_cs_add_buffer` as I/O function and (2) annotate the byte-swapping functions in Gallium. Overall, we had to add or change about 40 lines of source code, including annotations, in a single place. All our changes are published in the repository <https://rkapl.cz/repos/git/roman/mesa>. It contains the source code of Mesa augmented with our annotations and fixes for the endianness-related bugs in Radeon SI described below. For fixes of bugs found by Endicheck, we included the original Endicheck report in the commit message, under the ECNOTE header.

Figure 2 contains an example bug report produced by Endicheck with enabled origin tracking on Glxgears. The error report itself has three main parts (in this order): the problem description, origin stack trace (captured when the offending value is created) and point-of-check stack trace (recorded when some annotated I/O function is encountered). We show only fragments of stack traces for illustration (and to save space).

The problem description identifies the currently active thread, the nature of the error and the memory region containing the erroneous value. The memory region is identified by its address and an optional name provided by the program (“`radeon_add_buffer`” in

```

Thread 9 gallium_drv:0:
Memory does not contain data of Target endianness
Problem was found in block 0x41BF000 (named radeon_add_buffer)
at offset 0, size 8:
  0x41BF000: N N N N N N N N
The value was probably created at this point:
  at 0x8B787F7: si_init_msa_functions (si_state_msa.c:94)
  by 0x8B4F979: si_create_context (si_pipe.c:279)
  ...
  by 0x4C46661: glXCreateContext (glxcmds.c:427)
  by 0x10B67A: make_window.constprop.1 (glxgears.c:559)
  by 0x109A86: main (glxgears.c:777)
The endianness check was requested here:
  at 0x8B85C45: radeon_drm_cs_add_buffer (radeon_drm_cs.c:375)
  by 0x8B4A58B: si_set_constant_buffer (r600_cs.h:74)
  by 0x8B708D0: si_set_framebuffer_state (si_state.c:2934)
  ...
  by 0x55357FB: start_thread (pthread_create.c:465)
  by 0x5861B0E: clone (clone.S:95)

```

Fig. 2. Error report from Endicheck run on the Glxgears demo program

this case). Metadata are printed just for the part of the memory region that contains data with the wrong endianness, using this convention: N = native, U = undefined.

This particular error report (Figure 2) indicates that an array of floating-point values describing the *multisampling pattern* is not byte-swapped. Note that IEEE 754 floating point values also obey the endianness of the host platform, at least on the architectures x86, x64 and ARM. To repair the corresponding bug, we had to insert calls of byte-swapping functions at the code location where the floating-point array is produced.

During our experiments with Radeon SI and Glxgears, four endianness bugs in total were detected by Endicheck on the x86 architecture before testing on PowerPC. After we fixed the bugs, the Glxgears demo did successfully run. This shows that Endicheck detected all bugs it was supposed to and provided reports useful enough so that the bugs could be fixed. Here we also need to emphasize that the Glxgears demo, naturally, does not exercise all code in the Radeon SI driver, and fixing the whole driver would require lot of additional work.

Other programs. As we said at the beginning of this section, we evaluated Endicheck's ability to find endianness bugs and precision on a set of realistic programs. Our primary goal in this part of the evaluation was to assess the following aspects:

- the extent of annotations that is required for Endicheck to work properly,
- whether Endicheck is able to detect a bug in a given kind of programs, and
- how many false warnings are reported.

Before trying to answer these questions, we wanted to be sure that the subject programs contain endianness bugs. However, some of the programs that we considered

(OpenTTD, OpenArena and ImageMagick) are written in such a way that realistic endianness bugs cannot be injected into their codebase. ImageMagick uses a C++ abstraction layer for binary streams, which also handles endianness. OpenArena uses bit-oriented encoding for most parts of the network communication. OpenTTD uses an abstraction layer too, but the developer can still make an endianness-related mistake in certain cases, such as storing an array of `uint16_t` values as an array of `uint8_t` values. We manually injected synthetic endianness bugs into the code of all the programs where this was possible. In this process, we also annotated the byte-swapping functions (like `htonl`). The bugs were created by removing one usage of byte-swapping functions.

The results of experiments are summarized in Table 1. For each program, the table provides the following information: whether it was possible to analyze the program at all, whether some endianness bugs were found, overhead related to false warnings, and how many lines of source code were added or changed in relation to Endicheck annotations. Data for the Radeon SI driver are also included in the table for completeness.

Program	Analyzable	Injected bug	False positives	Actual bugs	Annotations
Radeon SI driver	✓ Yes	✓ Found	∅ Manageable (2)	✓ Found	cca 40 lines
BusyBox	✓ Yes	✓ Found	✓ No	None found	20 lines
OpenTTD	✓ Partially	✓ Found	∅ Manageable (2)	None found	59 lines
Ntpd	✓ Yes	✓ Found	✓ No	None found	1 line
X.Org	✓ Yes	✓ Found	✓ No	✓ Found	30 lines
OpenArena	∅ No				
ImageMagick	∅ No				

Table 1. Search for bugs: precision and necessary annotations

Data in Table 1 show that Endicheck could find the introduced bug in all the cases. Furthermore, Endicheck found two genuine endianness-related bugs in X.Org. The bugs were confirmed by the developers of X.Org and fixed in upstream³.

Endicheck also reports some false warnings, but their numbers are not overwhelming. Four cases in total occurred for the Radeon SI driver and OpenTTD (two in each). This is a manageable amount, which can be even suppressed using further annotations.

5.3 Performance

In this section, we report on the performance of Endicheck in terms of execution time overhead it introduces. We compare the performance data for programs instrumented with Endicheck, programs instrumented by the Memcheck plugin for Valgrind and programs without any instrumentation. For the purpose of experiments, we used the standardized benchmark SPEC CPU2000. Even though SPEC CPU2000 is a general benchmark, not tailored for endianness analysis, results of experiments with this benchmark

³ https://gitlab.freedesktop.org/search?group_id=&project_id=371&repository_ref=master&scope=commits&search=Roman+Kap1

indicate the performance of Endicheck when doing a real analysis, because the control-flow paths exercised within Endicheck and the Valgrind core during an experiment do not depend on the specific metadata (tag values).

We run all experiments on a T550 ThinkPad notebook with 12 GiB of RAM and an i5-5200 processor clocked at 2.20 GHz, under Arch Linux from Q2 2018. The SPEC2000 test harness was used for all the runs, with iteration count set to 3. We compiled both Memcheck and Endicheck by GCC v7.3.0 with default options. Note that we had to omit the benchmark program “gap”, because it produced invalid results when compiled with this version of GCC.

In the description of specific experiments, tables with results and their discussion, we use the following abbreviations:

- **EC**: Endicheck (valgrind `-tool=endicheck`)
- **MC**: Memcheck (valgrind `-tool=memcheck`)
- **-OT**: with precise origin tracking enabled (`-track-origins=yes`)
- **-IT**: with origin tracking enabled, but not fully precise (`-precise-origins=no`)
- **-P**: with memory protection enabled (`-protection=yes`)

Execution time. We divided our experiments designed for measuring the execution time into two groups. Our motivation was to ensure that all experiments, including the EC-OT configuration that incurs a large overhead, finish within a reasonable time limit. In the first group, we run the full range of configurations on the “test” data set provided by SPEC CPU2000, which is small compared to the full “reference” set, and used MC as the baseline for comparisons. Table 2 shows results for experiments in this group. All execution time data provided in this table are relative to MC, with the exception of data for the native configuration. The second group of experiments uses the full “reference” data set from SPEC CPU2000. Results for this group are provided in Table 3. In this case, we used the data for native (uninstrumented) programs as the baseline.

Program	Native (s)	MC (s)	MC-OT	EC	EC-P	EC-OT	EC-IT
bzip2	1.38	19.40	2.27x	2.07x	2.23x	33.87x	12.58x
crafty	0.70	18.70	2.21x	1.74x	1.78x	30.59x	11.07x
eon	0.09	6.60	1.73x	1.29x	1.34x	12.89x	4.23x
gcc	0.31	12.70	1.96x	1.92x	1.98x	24.17x	9.53x
gzip	0.47	6.29	2.11x	1.86x	1.97x	41.97x	14.96x
mcf	0.05	0.85	2.38x	1.27x	1.32x	11.88x	7.08x
parser	0.66	10.50	2.19x	2.13x	2.28x	41.24x	16.29x
perlbnk	4.31	5.52	1.10x	0.95x	0.95x	1.17x	1.05x
twolf	0.05	1.64	1.88x	1.16x	1.20x	14.09x	5.51x
vortex	1.06	56.90	2.23x	1.95x	2.04x	28.38x	9.86x
ypr	0.49	8.02	2.00x	1.70x	1.75x	22.94x	8.30x
G.mean	0.41	7.86	1.97x	1.59x	1.65x	18.17x	7.56x

Table 2. Execution times for the SPEC CPU2000 test data set, relative to Memcheck.

Program	Native (s)	MC	EC	EC-P
bzip2	66.3	11.63x	23.47x	24.45x
crafty	29.5	26.78x	48.10x	48.54x
eon	24.1	52.12x	93.36x	97.34x
gcc	27.8	27.73x	116.62x	122.48x
gzip	79.9	8.92x	15.93x	16.80x
mcf	67.10	2.71x	6.90x	6.94x
parser	89.9	10.78x	23.04x	23.86x
perlbmk	45.9	38.45x	93.62x	96.27x
twolf	93	12.43x	19.77x	19.52x
vortex	43.8	44.36x	91.03x	92.85x
vpr	54.7	10.49x	20.29x	20.68x
G.mean	51.29	16.59x	35.31x	36.25x

Table 3. Execution times for the SPEC CPU2000 reference data set, relative to native runs.

Data in Table 3 indicate that the average slowdown of Memcheck is by the factor of 16.59. Endicheck, in comparison, slows down the analyzed program by the factor of 35.31. This means Endicheck has roughly two times higher overhead than Memcheck with default options. According to data in Table 2, the same relative slowdown of Endicheck with respect to Memcheck is 1.65x. This difference between the results for the reference and test data sets is caused by the different ratio of the time spent instrumenting the code versus time spent running the instrumented code.

However, data in both tables also show that the performance of Endicheck with origin tracking is lacking compared to Memcheck with the same option. It was still usable for our Radeon SI OpenGL tests, but measurements indicate that there is a space for optimization. Nevertheless, certain relative slowdown between the configurations EC-OT and MC-OT probably cannot be avoided, because Endicheck must track origin information for much more data than Memcheck. Based on our experiments, we observed that creating the origin information is the most expensive operation involved. When the origin tags are created for each superblock, instead of every instruction, the execution times drop roughly by a factor of two (see the columns EC-OT and EC-IT).

5.4 Discussion

Based on the case study and results of experiments presented in the previous sections, we make the following general conclusions:

- Endicheck can find true endianness bugs in large real programs, assuming that the user correctly annotates all the byte-swapping functions and I/O functions.
- Using fairly complex metadata is feasible in terms of performance and encoding.
- Performance of Endicheck is practical even on large programs, despite the overhead and given that its current version is not yet optimized as well as Memcheck.
- Although Endicheck, due to precise dynamic analysis, requires less annotations to be specified manually than static analysis-based tools (e.g., Sparse), still it puts certain burden on the user.

Regarding the annotation burden, we already mentioned that the user has to carefully mark in particular all the I/O functions and byte-swapping functions, so that Endicheck can correctly update endianness tags associated with memory locations during the run of the analysis. While it would be possible to recognize byte-swapping functions automatically, e.g. by static code analysis, then the endianness analysis would have to be run on a machine with the native endianness different from the target endianness, so that actual byte-swaps will be present.

Another limitation of Endicheck from the practical perspective is handling of complex data transformations, a problem shared with taint analysis. The metadata cannot be correctly preserved through transformations such as encryption/decryption and compression/decompression. However, in many cases, the problem could be avoided by requiring an endianness check to be performed just before the respective transformation.

6 Related Work

As far as we know, the Sparse tool [11] used by Linux kernel developers, which we already mentioned, is the only one publicly available specialized tool tackling the problem of finding endianness bugs. The main advantage of Endicheck over Sparse is better precision in some cases, i.e. fewer false bug reports, since dynamic analysis, which observes actual program execution and runtime data values, is typically more precise than static analysis. Endicheck also does not require so many annotations of functions and variables as Sparse — when using Endicheck, typically just few places in the program source code need to be annotated manually. More specifically, Sparse expects that an input program code involves (i) the specialized bitwise data types (e.g., `_le32`) for all variables where endianness matters and (ii) the macros for conversion between regular types and bitwise types (e.g., `_le32_to_cpu`). With Endicheck, developers only have to annotate the byte-swapping functions used by the program (e.g., `htons` and `htonl` from the C library). On the other hand, Sparse has better coverage of program code, as it is based on static analysis.

The Valgrind dynamic analysis framework [6] comes bundled with a set of bug detection tools. Very popular is the Memcheck tool [5] for detecting memory access errors and leaks, which also served as an inspiration for the design and implementation of Endicheck. We mention the tool here, because it actually performs a variant of dynamic taint analysis — it marks each bit of the program memory as valid or invalid (tainted).

Closely related is also the runtime type checker Hobbes [2] for binary executables, which can detect some kinds of type mismatch bugs common in C programs. In order to reduce the number of false bug reports and to delimit integer values, Hobbes uses the mechanism of continuation markers — the first byte of each value has the marker unset, and the remaining bytes are set to indicate that they represent a continuation of an existing value. The analysis technique used by Hobbes could be modified to track endianness of integer values instead of distinguishing between pointers and integers, since one can model integers of different endianness as values that have different types (also like in the case of Sparse).

Another approach with functionality similar to Endicheck has been implemented within the LLVM/Clang plugin called DataFlowSanitizer [10]. It is a dynamic analysis

framework that (i) enables programs to define tags for data values and check for specific tags, both through its API functions, and (ii) propagates all tags with the data.

7 Conclusion

We have presented a new dynamic analysis tool, Endicheck, for detecting endianness bugs in C/C++ programs. The tool is built upon the Valgrind framework. Endicheck provides a useful, and in many settings also preferable, alternative to static analysis tools like Sparse, because (1) it reports quite precise results (i.e., a low number of false warnings) due to the nature of dynamic analysis and (2) requires less annotations (and other changes) in the source code of the subject program in order to be able to detect missing byte-swap operations. The results of our experimental evaluation show that Endicheck can (1) handle large complex programs and (2) identify actual endianness bugs, and it has practical performance overhead. Endicheck could also be used in automated testing scenarios, as a useful alternative to testing programs on both little- and big-endian processor architecture. A testing environment based on Endicheck might be easier to set-up than the environment based, for example, on virtual machines.

7.1 Future Work

Possible extensions of Endicheck, which could improve its precision and practical usefulness even further, include:

- More complex analysis approach based on explicit tagging of each byte in an integer data value with its position.
- Reporting arithmetic instructions that use data with target endianness.
- Automatically checking system calls such as `write` for correct endianness.
- Suppression files for endianness bug reports to eliminate false positives.

Another way to detect endianness bugs more precisely is to use comparative runs (i.e., a kind of equivalence checking). The key idea is to run a program on two machines, where one has a big-endian architecture and the other has a little-endian architecture, and compare the data leaving both variants of the program. This approach has the potential to be the most accurate, because it can even detect problems in cases when data leaving the program are encrypted or compressed. On the other hand, it cannot always detect situations when the program forgets to byte-swap input data, unless the error affects one of the output values with *concrete* endianness.

Acknowledgments. This work was partially supported by the Czech Science Foundation project 18-17403S and partially supported by the Charles University institutional funding project SVV 260451.

References

1. Bond, M.D., Nethercote, N., Kent, S.W., Guyer, S.Z., McKinley, K.S.: Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In: Proceedings of OOPSLA 2007. ACM (2007)

2. Burrows, M., Freund, S.N., Wiener, J.L.: Run-Time Type Checking for Binary Programs. In: Proceedings of CC 2003. LNCS, vol. 2622. Springer (2003)
3. Kápl, R.: Dynamic Analysis for Finding Endianness Bugs. Master thesis, Charles University, Prague, June 2018.
4. Liu, Y., Milanova, A.: Static Analysis for Inference of Explicit Information Flow. In: Proceedings of PASTE 2008. ACM (2008)
5. Seward, J., Nethercote, N.: Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In: Proceedings of USENIX 2005 Annual Technical Conference. USENIX Association (2005)
6. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proceedings of PLDI 2007. ACM (2007)
7. Nethercote, N., Seward, J.: How to Shadow Every Byte of Memory Used by a Program. In: Proceedings of VEE 2007. ACM (2007)
8. Newsome, J., Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In: Proceedings of NDSS 2005. The Internet Society (2005)
9. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker. In: Proceedings of USENIX 2012 Annual Technical Conference. USENIX Association (2012)
10. Clang 8 documentation / DataFlowSanitizer. <https://clang.llvm.org/docs/DataFlowSanitizer.html> (accessed in October 2019)
11. Sparse: a semantic parser for C programs. <https://lwn.net/Articles/689907/> (accessed in October 2019)

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

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





Describing and Simulating Concurrent Quantum Systems

Richard Bornat^{1,4}, Jaap Boender^{2,5}, Florian Kammüller^{1,6}, Guillaume Poly^{3,7}, and Rajagopal Nagarajan^{1,8}

¹ Department of Computer Science, Middlesex University, London, UK

² Hensoldt Cyber GmbH, Taufkirchen, Germany

³ Widmee, Région de Bordeaux, France

⁴ R.Bornat@mdx.ac.uk

⁵ jacob.boender@hensoldt-cyber.com

⁶ F.Kammüller@mdx.ac.uk

⁷ guillaume.gwigwi.poly@gmail.com

⁸ R.Nagarajan@mdx.ac.uk



Abstract. We present a programming language for describing and analysing concurrent quantum systems. We have an interpreter for programs in the language, using a symbolic rather than a numeric calculator, and we give its performance on examples from quantum communication and cryptography.

Quantum cryptographic protocols such as BB84 QKD [3] and E92 QKD [7] offer unconditional statistical security. These protocols have been implemented in commercial products; various QKD networks have been built around the world; and China has launched a dedicated satellite for quantum communication. The security of the protocols has been established information-theoretically, but their implementations may have security loopholes. We intend to investigate the security question, eventually by using formal methods to verify the properties of implementations, but first by simulation of protocols expressed as programs.

Large companies are developing full-stack solutions for implementing quantum algorithms, and quantum computers will likely be network-linked. Although we have focused on quantum communication and cryptography protocols, aspects of our work will be applicable to distributed quantum computation.

Concurrent quantum systems, such as communication and cryptographic protocols assume physically-separated *agents* (Alice, Bob, etc.) who communicate by sending each other *qubits* (quantum bits: polarised photons, for example) and classical bit-strings. There are a few dedicated, high-level programming languages for quantum systems such as Microsoft's Q# [2]. They focus on single-machine computation and lack a treatment of communication, but a protocol simulation must ensure, for example, that a qubit transferred from one agent to another can't be used again by the sender and can't be used by the receiver before it is sent. We decided therefore to take a process-calculus approach, and we have implemented a tool inspired by CQP [9]. Our implementation is called qtpi [1], and uses symbolic rather than numeric quantum calculation. Programs

© The Author(s) 2020

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12079, pp. 271–277, 2020.

https://doi.org/10.1007/978-3-030-45237-7_16

are checked statically, before they run, to ensure that they obey real-world restrictions on the use of qubits (no cloning, no sharing). Unlike CQP, which preserves all possible outcomes, labelling each with a probability, qtpi takes a single execution path, making probabilistic choices between outcomes.

We have used qtpi to simulate simple protocols such as teleportation, and some more involved ones including the quantum key-distribution protocols BB84 [3] and E92 [7]. Each of these involves transmission of qubits and public transmission of classical messages (in the case of BB84, over an authenticated channel [13]), all of which is simulated. It is early days in our development of the tool, so there is as yet no provision for formal proof, but in these examples we can already simulate well over 1M qubit transfers per minute on a small laptop – i.e. we can simulate largish examples in a useful time.

1 Processes

Protocols are carried out by *agents* which send each other messages but share no other information. We simulate agents by *processes* which share no data or variables. Typical protocol steps from the literature are

- obtain a qubit, perhaps initialised to one of $|0\rangle$, $|1\rangle$, $|+\rangle$ or $|-\rangle$;
- put a qubit through a gate such as I, H, X, etc.;
- measure a qubit;
- send or receive a qubit;
- send or receive a classical value, such as a list of numbers or bits.

In addition an agent may perform a calculation, such as generating 1000 random bits or encrypting/decrypting a message or checking the values received in a message. Calculations aren't protocol steps and don't affect qubit state, though they often depend on the results of measuring qubits and their results often influence subsequent protocol steps. Our processes have analogues of protocol steps and calculations. In addition we are able to create processes, to choose conditionally between different processes and to set up a collection of processes running simultaneously.

The aim of our work is to mathematically analyse programs which describe quantum systems. Towards that end we have a semantics of quantum-mechanical calculation [5], written in Coq [10]. That is work in progress: for the time being we are able to execute our protocol-programs using our simulator [1].

1.1 A programming language

Our language has two distinct notations: a protocol-step language, which is derived from the pi-calculus [11], and a functional calculation language, somewhat in the style of Miranda [12]. Neither language has assignment, although qubit measurement does change program state and so needs special attention. The protocol-step language has recursion, but only tail recursion: i.e. nothing can follow a process invocation step (but note that parallel execution of sub-processes provides more complexity).

Following the pi-calculus we use *channels* to communicate between processes. So Alice doesn't send to Bob, she sends down a channel which Bob can read from – or perhaps it might be Eve, if there is interference. Channels are values, so you can set up communication between two processes by giving them the same channel-argument when you create them, and you can send channel values in messages to alter connections dynamically.

In the protocol-step language steps are separated by dots (‘.’) and choices are made between processes rather than single or multiple steps. Channels are created by (new c); send is $C!E, \dots, E$; receive is $C?(x, \dots, x)$; qubits are created by (newq q); quantum gating is $Q, \dots, Q \gg G$; quantum measurement $Q-/- (x)$.

In the expression language there is function application ($f \text{ arg}$), arithmetic and Boolean calculation, conditional choice and recursion. It uses infinite-precision rationals for numerical calculations.

1.2 Symbolic quantum calculation

Quantum calculations can be described using *quantum circuits*: diagrams such as Fig. 1 show how qubits (one per line) are put through gates (boxes, line-connectors) and/or measured (meter symbols) giving a classical 0/1 result.

In quantum mechanics the state of a qubit is a vector $a|0\rangle + b|1\rangle$, with $|a|^2 + |b|^2 = 1$. Here $|0\rangle$ and $|1\rangle$ are the computational basis vectors, a and b are complex amplitudes, and $|a|^2$ and $|b|^2$ give the probability of measuring the state as $|0\rangle$ or $|1\rangle$. In qtpi a single isolated qubit is therefore a pair of complex numbers, and quantum gates, such as the H, X and Z gates of Fig. 1, are square matrices of complex numbers which modify the state by multiplication. The state of n entangled qubits is a 2^n -element vector, matrices which manipulate all of it have to be $2^n \times 2^n$, so calculations with large entanglements can rapidly grow out of the range of straightforward simulation. Luckily, quantum security protocols typically work with a small number of qubits at a time.

Because our calculations are simple, we can afford to implement them symbolically. We use h for $\sqrt{1/2}$; it is also equal to $\sin(\pi/4)$ and $\cos(\pi/4)$. A great deal of formulae can be expressed in terms of powers of h : for example $\cos(\pi/8) = \sqrt{(1+h)/2}$.

Symbolic calculation involves lots of symbolic simplification. That makes it relatively slow, compared to calculation with floating-point numbers, but it is absolutely accurate – $h^2 + h^2$, for example, is exactly 1. When measuring, we must convert symbolic probabilities into numbers. But that is part of a statistical calculation, so minor inaccuracy is acceptable.

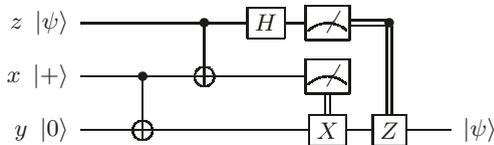


Fig. 1. Quantum circuit for teleportation

```

proc System () =
  (newq x=|+>, y=|0>) x,y>>CNot .
  (new c:^bit*bit) | Alice(x,c) | Bob(y,c)

proc Alice (x:qbit, c:^bit*bit) =
  (newq z)
  out!["initially Alice's z is "] . outq!(qval z) . out!["\n"] .
  z,x>>CNot . z>>H . z-/-(vz) . x-/-(vx) . c!vz,vx . _0

proc Bob(y:qbit, c:^bit*bit) =
  c?(b1,b2) .
  y >> match b1,b2 . + 0b0,0b0 . I
                + 0b0,0b1 . X
                + 0b1,0b0 . Z
                + 0b1,0b1 . Z*X .
  out!["finally Bob's y is "] . outq!(qval y) . out!["\n"] . _0

```

Fig. 2. Teleportation of an unknown quantum state, with logging

1.3 No cloning

In the real quantum world there is no way of cloning a qubit – you can't start with a qubit in some arbitrary state and finish up with two qubits in that state. That, plus the fact that measurement irrevocably alters a qubit's state, is what provides quantum security protocols with unconditional security – though the uncertainty of measurement means that the guarantee is probabilistic, not absolute. A programming language which simulates quantum effects should therefore not allow copying of the value of a qubit variable. We use language restrictions to facilitate anti-cloning checks: in particular we severely restrict the use of qubits in data structures, in messages, and after measurement or transmission. Those checks are partly implemented by typechecking, partly by an efficient static symbolic execution before simulation begins.

1.4 Other notable features

Randomised priority queues of runnable processes and waiting communication offers ensure non-deterministic execution, and are used to eliminate infinite unfairness. Logging steps can be pushed into subprocesses to clarify protocol descriptions, leaving a marker in the logged process to show where it should occur (see examples in artifact [6]). Type descriptions are almost entirely optional.

2 Straightforward description

Our aim is to provide a programming language in which protocol descriptions are transparently easy to read. For example, Fig. 2 shows teleportation [4] using three processes: Alice and Bob carry out the protocol, and System sets up the

communication between them. The calculation follows the circuit in Fig. 1, but is shared between agents obeying the anti-cloning restrictions.

The System process creates qubits x and y (`newq ..`), initialised to $|+\rangle$ and $|0\rangle$, and entangles them using a CNot gate (`x,y>> ..`). It creates a channel c which carries pairs of bits (`new c ..`), and then splits into two subprocesses: one becomes Alice, taking one of the qubits and the channel; the other becomes Bob, with the other qubit and the same channel. Those processes run in parallel.

The Alice process creates a new qubit z , *without specifying its state*, and logs that state (the anti-cloning restrictions make this tricky). Then it puts z and x through a CNot gate (`z,x>> ..`), puts z alone through a Hadamard gate (`z>>H`), and finally measures first z (`z-/- (vz)`), then x (`x-/- (vx)`), giving bits vz and vx . Finally it sends those bits to Bob on the c channel (`c!...`). The overall effect is subtle, because first System's actions entangle x and y , so that measurement of x constrains y , and then Alice entangles z , x and y , so that measurement of z constrains both x and y .

The Bob process waits to receive Alice's message (`c? ..`), and calculates a gate (`match ..`) to process the results depending on one of four possibilities for the two bits it receives (note one of the gates is the matrix product of Z and X). It puts y through that gate (`y>> ..`) and logs the result. The output of this program is always

```
initially Alice's z is 2:(a2|0>+b2|1>)
finally Bob's y is 1:(a2|0>+b2|1>)
```

where a_2 and b_2 are unknown symbolic amplitudes. A sample execution trace, edited for brevity, shows the states produced by Alice's actions: qubit 0 is x , 1 is y , 2 is z ; initially 0 and 1 are entangled, and the first step entangles all three.

```
Alice 2: (a2|0>+b2|1>), 0: [0;1] (h|00>+h|11>)) >> Cnot;
  result 2: [2;0;1] (h*a2|000>+h*a2|011>+h*b2|101>+h*b2|110>),
    0: [2;0;1] (h*a2|000>+h*a2|011>+h*b2|101>+h*b2|110>))
Alice 2: [2;0;1] (h*a2|000>+h*a2|011>+h*b2|101>+h*b2|110>) >> H;
  result 2: [2;0;1]
    (h(2)*a2|000>+h(2)*b2|001>+h(2)*b2|010>+h(2)*a2|011>
    +h(2)*a2|100>-h(2)*b2|101>-h(2)*b2|110>+h(2)*a2|111>)
Alice: 2: (.. as above ..) -/- ;
  result 0 and 0: [0;1] (h*a2|00>+h*b2|01>+h*b2|10>+h*a2|11>),
    1: [0;1] (h*a2|00>+h*b2|01>+h*b2|10>+h*a2|11>))
Alice: 0: [0;1] (h*a2|00>+h*b2|01>+h*b2|10>+h*a2|11>) -/- ;
  result 1 and 1: (b2|0>+a2|1>)
Chan 2: Alice -> Bob (0,1)
Bob 1: (b2|0>+a2|1>) >> X; result 1: (a2|0>+b2|1>)
```

Tracing several executions shows that Alice's measurements don't always give the same results in vz , vx and qubit 1, so Bob doesn't always use the same gate(s). The qubit z is never sent in a message, is destroyed by Alice's measurement, and its amplitudes are unknown to the program, but y always finishes up in the state that z began in. Without symbolic calculation we couldn't do such a simulation.

3 Performance on examples

We can run various simulations of the quantum key-distribution protocol BB84 [3], with Alice and Bob and various Eve processes. In order to generate a one-time key to encrypt an n -bit message, Alice needs to send many more bits than n , and our simulation allows us to experiment with various parameters of her calculation to see what happens. Here is a shortened display of part of the output of an example simulation (timing measurements made on VirtualBox Ubuntu 18.10, on a 7-year-old MacBook Air with 8GB RAM):

```
length of message? 4000; length of a hash key? 40;
minimum number of checkbits? 500; number of sigmas? 10;
number of trials? 100
```

```
13718 qubits per trial; 0 interfered with; 100 succeeded
```

It takes about 0.6 seconds for each trial, but overall it makes 1.3M qubit transfers and measurements in 60 CPU seconds. With an intercept-and-resend Eve, the same exchanges take 95 seconds, but Eve’s interference is detected every time. With a very short message and very few checkbits we can show that even such a naive Eve can sometimes win, as statistical analysis predicts.

Our simulation of E92 QKD [7] uses 20 000 entangled qubit pairs per trial for the same-sized problem. Because the protocol calculations are more complicated and our calculation language is interpreted rather than compiled, simulation takes over 4 CPU minutes.

Qtqi can handle larger entanglements. In about 13 seconds it’s able to set up and measure one ‘brick’ (ten qubits, all CZ-entangled) of the measurement-based quantum computing mechanism in [8] – but that’s too small to be useful, and larger entanglements are exponentially worse.

4 Conclusions

We have a quantum programming language which allows description of protocols with multiple agents. It has protection, built from well-understood computer science foundations, against cloning of qubits within a simulation. It is not yet able to deal efficiently with entanglements of more than a few qubits. Its symbolic calculator is fast enough for the protocols we have examined.

5 Data Availability and Acknowledgements

The qtqi interpreter and the examples referred to in the paper are available at <https://doi.org/10.6084/m9.figshare.11882592>. Our research was supported by UK National Cyber Security Centre through the VeTSS project “Formal Verification of Quantum Security Protocols using Coq”. Nagarajan was also supported by EU Cost Action IC1405 “Reversible Computation - Extending Horizons of Computing”. We thank Simon Gay for helpful discussions.

References

1. Qtpi protocol simulator, <https://github.com/mdxtoc/qtpi>, accessed on 2020.02.13
2. The Q# Programming Language, <https://docs.microsoft.com/en-us/quantum/quantum-qr-intro>, accessed on 2020.02.13
3. Bennett, C.H., Brassard, G.: Quantum cryptography: Public key distribution and coin tossing. In: Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing. p. 175. India (1984)
4. 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. *Physical Review Letters* **70**(13) (1993)
5. Boender, J., Kammüller, F., Nagarajan, R.: Formalization of quantum protocols using Coq. In: The 12th International Workshop on Quantum Physics and Logic. vol. 195, pp. 71–83 (2015). <https://doi.org/10.4204/EPTCS.195.6>
6. Bornat, R., Boender, J., Kammüller, F., Poly, G., Nagarajan, R.: Figshare (2020), <https://doi.org/10.6084/m9.figshare.11882592>, visited 2020/02/21
7. Ekert, A.K., Rarity, J.G., Tapster, P.R., Massimo Palma, G.: Practical quantum cryptography based on two-photon interferometry. *Phys. Rev. Lett.* **69**, 1293–1295 (Aug 1992). <https://doi.org/10.1103/PhysRevLett.69.1293>
8. Ferracin, S., Kapourniotis, T., Datta, A.: Reducing resources for verification of quantum computations. *Physical Review A* **98**(2), 022323 (2018)
9. Gay, S.J., Nagarajan, R.: Communicating quantum processes. In: 32nd Symposium on Principles of Programming Languages (POPL 2005). pp. 145–157 (2005). <https://doi.org/10.1145/1040305.1040318>, also arXiv:quant-ph/0409052
10. INRIA: The Coq Proof Assistant, <https://coq.inria.fr>, accessed on 2020.02.13
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. *Inf. Comput.* **100**(1), 1–40 (1992)
12. Turner, D.A.: Miranda: a non-strict functional language with polymorphic types. In: Proc. of a conference on Functional programming languages and computer architecture. pp. 1–16. Springer-Verlag New York, Inc., New York, NY, USA (1985)
13. Wegman, M.N., Carter, J.L.: New hash functions and their use in authentication and set equality. *Journal of computer and system sciences* **22**(3), 265–279 (1981)

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

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





EMTST: Engineering the Meta-theory of Session Types

David Castro¹, Francisco Ferreira², and Nobuko Yoshida¹

Imperial College London,
{d.castro-perez, f.ferreira-ruiz, n.yoshida}
@imperial.ac.uk



Abstract Session types provide a principled programming discipline for structured interactions. They represent a wide spectrum of type-systems for concurrency. Their type safety is thus extremely important. EMTST is a tool to aid in representing and validating theorems about session types in the Coq proof assistant. On paper, these proofs are often tricky, and error prone. In proof assistants, they are typically long and difficult to prove. In this work, we propose a library that helps validate the theory of session types calculi in proof assistants. As a case study, we study two of the most used binary session types systems: we show the impossibility of representing the first system in α -equivalent representations, and we prove type preservation for the revisited system. We develop our tool in the Coq proof assistant, using locally nameless for binders and small scale reflection to simplify the handling of linear typing environments.

Keywords: Concurrency · proof assistants · meta-theory · session-types.

1 Introduction

Given the prevalence of distributed computing and multi-core processors, concurrency is a key aspect of modern computing. The transition from sequential models of computation to concurrent systems has huge practical and theoretical consequences. Message passing calculi (like the π -calculus) have been used to model these systems since their introduction by Milner et al. [15]. Notably, in many cases *typing disciplines* are used as a way to control concurrent and distributed behaviour. Certifying basic typed π -calculi is important for both the safety of implementations and the trustworthiness of new theories.

In this work, we concentrate on providing tools for reasoning about *session types* [10], a typing discipline for structured interactions in distributed systems. Session types are applied to a wide range of problems, and their properties, such as deadlock-freedom, are well studied. These calculi are very expressive, and rather complex, with features like: shared and linear communication channels, name passing, and fresh name generation. Given this complexity, it is not surprising that some innocent looking extensions violated the type safety properties of the calculus in several literature (as pointed out by [23]). In consequence, the

interest for mechanisation and formal proofs has risen significantly as a means to increase the trust on systems.

Type systems offer certain security properties by construction. These guarantees are backed by rigorous proofs (these proofs conform the meta-theory of the system). Moreover, these proofs are cumbersome to write, maintain and extend. Proof assistants aim to help with these problems. In this work, we develop the EMTST library to aid in the implementation of session calculi type systems. As a form of validation, we implement and replicate results in the meta-theory of binary session types. Concretely, we use the Coq proof assistant [20] to study the representation and meta-theory of the two systems described in [23].

EMTST uses *locally nameless* (LN)[1, 5] variable binders to represent syntax. The tool implements a LN library with extended support for multiple binding scopes, a robust environment implementation suitable for the challenges of session typing disciplines. The library and lemmas are written taking advantage of boolean reflection through the use of the `Ssreflect` [7] library.

We implement two case studies from [23]. The first study that we refer to as *the original system* and the second that we refer to as *the revised systems*. Notably, the way the original system handles names (in Sect. 3.1), makes its representation impossible when using intrinsically α -convertible terms (e.g: locally nameless, de Bruijn indices, and many others). Furthermore in Sect. 3.2, we discuss how the revised system allows us to implement and prove type preservation. In hindsight, this problem appears as evident, but it is an unexpected consequence, and it shows that mechanising proofs brings further understanding even to well-established and thoroughly studied systems. EMTST and our case studies are available at <https://github.com/emtst/emtst-proof>.

The rest of the paper is structured in the following way: in the next section we introduce the ideas and design behind EMTST our library for mechanising the meta-theory of session types. Subsequently in Sect. 3, we present the two case studies: in Sect. 3.1 the original system from [23, 11] and the revisited system in Sect. 3.2. We finalise, by giving a conclusion and related work.

2 EMTST: a Tool for Representing the Meta-theory of Session Types

The study of meta-theory (i.e: proving a system has the expected properties) gives us confidence in the design. Additionally, proof formalisations, not only give us confidence in the results, but also often result in new insights about a problem. This is due to the fact that successful mechanisations require very precise specifications and careful thought to define and revisit all the concepts. In this context, EMTST is a tool that implements locally nameless (initially proposed by [8, 14, 13], and more recently further developed in [1, 5]) with multiple binding scopes, and a robust typing environment implementation using boolean reflection (by building on top of `ssreflect` [7]).

The key concept of LN is to use de Bruijn indices [2] for bound variables and names (sometimes called “atoms” in the literature) for free variables. A

representation of syntax is well formed, namely *locally closed*, when this invariant is respected (i.e.: no de Bruijn index is free). Finally, in order to deal with open terms, there are two convenient operations on syntax, one is to *open* binders in terms, and one to *close* binders. The former substitutes a bound variable with a fresh name, and the other does the converse. For more details, refer to our tech report [4], the references, and the implementation.

2.1 Environments and Multiple Name Scopes

```

Module Type ATOM.
  Parameter atom : Set.
  Definition t := atom.

  (* atoms can be compared to booleans *)
  Parameter eq_atom : atom → atom → bool.
  Parameter eq_reflect : ∀ (a b : atom),
    ssrbool.reflect (a = b) (eq_atom a b).
  Parameter atom_eqMixin : Equality.mixin_of atom.
  Canonical atom_eqType := EqType atom atom_eqMixin.

  Parameter fresh : seq atom → atom.
  Parameter fresh_not_in : ∀ l, (fresh l) ∉ l.
  (* ... *)
End ATOM.

```

Figure 1. The type of atoms

and channel names), and the final one (`theories/Env.v`) implements contexts and typings as finite maps, with emphasis on supporting the linearity requirements of various session typing disciplines.

We use module types and parametrised modules to abstract the type of atoms together with their supported operations. Figure 1 shows the interface for working with atoms: how to compare them and functions to obtain a fresh atom given a finite sequence of atoms (definition: `fresh`), and to have proof that the fresh atom is actually fresh (definition: `fresh_not_in`).

Environments. Environments are parametrised over two types, one for the keys, and one for the type of values. Environments `env` are either undefined, or a finite map of unique keys and values. All the operations keep the invariant that any operation that would lead to a duplicated entry key makes the tree undefined. We define the expected operations and lemmas over the type `env`. We provide an extensive library of proved theorems about environments that is tailored to support linear and affine systems.

EMTST is used in the two formalisations in Sect. 3.1 and 3.2 and we claim they are also suitable for other mechanisations where resource sensitivity and locally nameless are required. A release version of EMTST is available at [3] and the public repository at: <https://github.com/emtst/emtst-proof>.

3 Two Case Studies on Binary Session Types

EMTST is intended to help with the complex binding structure of concurrent calculi that have names as a first class notion together with linear or affine typing

Locally nameless implementation is in three files. The first (`theories/Atom.v`) provides the basic definition and specification of atoms to act as names, the second one (`theories/AtomScopes.v`) provides a way to create multiple disjoint sets of names for representing variables in the different scopes that session types require (e.g. variables

disciplines. We study two seminal session type systems in the literature. First the *original system*, from Honda, Vasconcelos and Kubo’s binary session type system [11] that is a milestone in the development of type systems for concurrent process calculi. This system types structured interaction between processes and supports channel mobility, that is higher-order sessions. Second, we implement the revisited session type presentation from [23], inspired by [6]. Our technical report [4] contains an extensive presentation.

3.1 The Original System

Process $P, Q, R ::=$			
request $a(k).P$	session request	if e then P else Q	conditional
accept $a(k).P$	session accept	$P \mid Q$	parallel
$k![e]; P$	data send	inact	inaction
$k?(x).P$	data receive	$\nu_n(a).P$	name hiding
$k \triangleleft m; P$	selection	$\nu_c(k).P$	channel hiding
$k \triangleright \{1 : P[r : Q]\}$	branching	$!P$	replication
throw $k[k']; P$	channel send		
catch $k(k').P$	channel receive		
$e ::= \mathbf{true} \mid \mathbf{false} \mid \dots$	expression	$m ::= 1 \mid r$	labels

Figure 2. Syntax using names

Figure 2 presents the syntax following [23], where names are ranged by a, b, c, \dots , channels are ranged by k and k' . Notice that all the places where there are variable binders are denoted with parenthesis followed by a dot (e.g: $k?(x).P$). The syntax is straightforwardly defined as the **proc** inductive type in **theories/Syntax0.v** and following the LN technique the locally closed predicate, that formalises the binding structure, is defined as the predicate **lc**.

Besides its syntax, the original system is specified by its reduction, congruence and typing relations. We want to call attention to an important reduction rule for passing names:

$$[\text{PASS-NM}] \quad \mathbf{throw} \ k \ [k']; P \mid \mathbf{catch} \ k \ (k').Q \longrightarrow P \mid Q$$

This rule states that when passing a channel k' the receiving end has to bind a channel using the same name (or be α -convertible to that name). Notoriously, the name k' is a bound name in the receiving end, and the restriction imposed by the rule is a subtle change to the equality up-to α -conversion convention. Moreover, relaxations of that requirement may break subject reduction, a complete discussion is presented in Sect. 3 of [23]. As it is, this rule cannot be formalised in a representation that cannot distinguish between α -equivalent terms. Since in these representations, one cannot talk about the actual name of a bound variable. This is fundamentally what it means to be *up-to α -equality*. As a consequence, in locally nameless we are forced to specify the following rule:

$$[\text{PASS-LN}] \quad \frac{\mathbf{lc} \ P \quad \mathbf{body} \ Q}{\mathbf{throw} \ k \ [k']; P \mid \mathbf{catch} \ k \ ().Q \longrightarrow P \mid Q^{k'}}$$

In this version of the rule, the bound name is just an anonymous de Bruijn index, and when it is opened it is assigned the same name k' . This change might look innocent, but it breaks subject reduction. In `theories/Types0.v`, we show that the same counter example from [23] is typable and that it breaks subject reduction. This is presented in the `CounterExample` module and in the `oft_reduced` lemma. In the next section, we discuss how this problem was addressed.

3.2 The Revised System

As discussed in Sect. 3.1 and [23], the presentation of the original session types calculus [11] makes extending it (and representing it in LN) a delicate operation. Fortunately, the revised system (also from [23], inspired by [6]) proposes a solution. Indeed, this solution is readily implementable using LN (and many other representations with implicit α -equivalence).

The key insight in the design of the revisited system is considering *channel endpoints* instead of just *channels*. As before, a new channel is created when a requested session is accepted, and each continuation gets one of the *endpoints* of the newly created channel.

<pre style="margin: 0;"> Inductive proc : Set := request : scvar → <u>proc</u> → proc accept : scvar → <u>proc</u> → proc send : channel → exp → proc → proc receive: channel → <u>proc</u> → proc select : channel → label → proc → proc branch : channel → proc → proc → proc throw : channel → channel → proc → proc catch : channel → <u>proc</u> → proc ife : exp → proc → proc → proc par : proc → proc → proc inact : proc nu_ch : <u>proc</u> → proc (* hides a channel name *) nu_nm : <u>proc</u> → proc (* hides a name *) bang : proc → proc (* process replication *) </pre>	<p>Legend:</p> <hr style="border: 0.5px solid black;"/> <table style="border-collapse: collapse; width: 100%;"> <tr> <td style="padding-right: 10px;"><u>proc</u></td> <td>process binds variable from \mathbb{A}_{SC}</td> </tr> <tr> <td style="padding-right: 10px;"><u>proc</u></td> <td>process binds variable from \mathbb{A}_{EV}</td> </tr> <tr> <td style="padding-right: 10px;"><u>proc</u></td> <td>process binds variable from \mathbb{A}_{LC}</td> </tr> <tr> <td style="padding-right: 10px;"><u>proc</u></td> <td>process binds variable from \mathbb{A}_{CN}</td> </tr> </table>	<u>proc</u>	process binds variable from \mathbb{A}_{SC}	<u>proc</u>	process binds variable from \mathbb{A}_{EV}	<u>proc</u>	process binds variable from \mathbb{A}_{LC}	<u>proc</u>	process binds variable from \mathbb{A}_{CN}
<u>proc</u>	process binds variable from \mathbb{A}_{SC}								
<u>proc</u>	process binds variable from \mathbb{A}_{EV}								
<u>proc</u>	process binds variable from \mathbb{A}_{LC}								
<u>proc</u>	process binds variable from \mathbb{A}_{CN}								

Figure 3. Syntax representation annotated with binders

For the revisited system's formalisation we distinguish binders in four categories (as shown in Figure 3): First, expression variables, with names from the set \mathbb{A}_{EV} , then shared channel variables from \mathbb{A}_{SC} , also linear channel variables from \mathbb{A}_{LC} , and finally channel names from \mathbb{A}_{CN} (these names can also be bound in restrictions). Channel names are not variables, but objects that exist at runtime.

Multiple disjoint sets of names simplify reasoning about free names (concretely, it avoids freshness problems among different kinds of binders). This is an engineering compromise, as having more binders duplicates some easy theorems but, in exchange, they simplify the harder theorems that rely on facts about LN open/close operations. Other compromises are possible.

This concludes the technical development, and represents a full proof of subject reduction for binary types, following the revised system¹ as defined in [23].

4 Related Work and Conclusions

We presented EMTST, a tool conceived to aid in the mechanisation of session calculi. Our tool supports locally nameless representations with many disjoint atom scopes, and a versatile representation of environments. All while taking advantage of the small scale reflection style of proofs. We validated our design by formalising the subject reduction proof for a full session calculus type system. And, we explored issues with adequacy when, for example, systems contain fragile specifications.

Tools like Metalib [22] (implemented based on [1]) and AutoSubst [18] exist, but lack the ability to represent different binding scopes in the same syntax. Also, Polonowski [17] implements a library for generic environments, while this library is similar to ours, it does not make use of boolean reflection, that, in our opinion simplifies dealing with the equality of environments. While these libraries were influential, our requirements of multiple scopes of binding and boolean reflection proofs, means that we needed to develop EMTST, our own fit for purpose library.

Finally, formalisations of session types in proof assistants exist in the literature (e.g.: [21, 24, 19, 16, 9]). Most of them with ad-hoc binder representations. They are not necessarily meant to be reused or general enough for other developments. This paper, and the EMTST library are a step towards helping this become easier. For that purpose we developed the library and validated its claims by formalising existing systems from the literature. In the process (see Sect. 3.1 vs Sect. 3.2), we motivate how early mechanisation would help avoid problems in the presentation of a system. In the future, we plan to extend our use of the library to reason about multiparty session types [12] and other systems.

Acknowledgements

This work was supported in part by EPSRC projects EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, and EP/T006544/1.

¹ A minor difference is that we use a simpler version of recursion compared to the original paper.

Bibliography

- [1] Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 3–15. POPL '08, ACM, New York, NY, USA (2008)
- [2] de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math* 34(5), 381–392 (1972)
- [3] Castro, D., Ferreira, F., Yoshida, N.: EMTST - Engineering Meta-theory of Session Types (Oct 2019), <https://doi.org/10.5281/zenodo.3516299>
- [4] Castro, D., Ferreira, F., Yoshida, N.: Engineering the meta-theory of session types. Tech. Rep. 2019/4, Imperial College London (2019), <https://www.doc.ic.ac.uk/research/technicalreports/2019/#4>
- [5] Charguéraud, A.: The locally nameless representation. *Journal of Automated Reasoning* 49(3), 363–408 (Oct 2012)
- [6] Gay, S., Hole, M.: Subtyping for session types in the pi calculus. *Acta Informatica* 42(2), 191–225 (Nov 2005)
- [7] Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in coq. *Journal of Formalized Reasoning* 3(2), 95–152 (2010)
- [8] Gordon, A.D.: A mechanisation of name-carrying syntax up to alpha-conversion. In: Joyce, J.J., Seger, C.J.H. (eds.) *Higher Order Logic Theorem Proving and Its Applications*. pp. 413–425. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
- [9] Goto, M., Jagadeesan, R., Jeffrey, A., Pitchar, C., Riely, J.: An extensible approach to session polymorphism. *Mathematical Structures in Computer Science* 26(3), 465–509 (2016)
- [10] Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) *CONCUR'93*. pp. 509–523. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
- [11] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) *Programming Languages and Systems*. pp. 122–138. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
- [12] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. of 35th Symp. on Princ. of Prog. Lang. pp. 273–284. POPL '08, ACM, New York, NY, USA (2008)
- [13] McBride, C., McKinna, J.: Functional pearl: I am not a number–i am a free variable. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. pp. 1–9. Haskell '04, ACM, New York, NY, USA (2004)
- [14] McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. *Journal of Automated Reasoning* 23(3), 373–409 (Nov 1999)
- [15] Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. *Info.& Comp.* 100(1) (1992)

- [16] Orchard, D.A., Yoshida, N.: Using session types as an effect system. In: Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015. pp. 1–13 (2015)
- [17] Polonowski, E.: Generic environments in coq. CoRR abs/1112.1316 (2011), <http://arxiv.org/abs/1112.1316>
- [18] Schäfer, S., Tebbi, T., Smolka, G.: Autosubst: Reasoning with de bruijn terms and parallel substitutions. In: Zhang, X., Urban, C. (eds.) Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015. LNAI, Springer-Verlag (Aug 2015)
- [19] Tassarotti, J., Jung, R., Harper, R.: A higher-order logic for concurrent termination-preserving refinement. In: Yang, H. (ed.) Programming Languages and Systems. pp. 909–936. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
- [20] The Coq Development Team: The Coq Proof Assistant Reference Manual v. 8.6.1. Institut National de Recherche en Informatique et en Automatique (2016)
- [21] Thiemann, P.: Intrinsically-typed mechanized semantics for session types. In: Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019. pp. 19:1–19:15. PPDP '19, ACM, New York, NY, USA (2019)
- [22] Weirich, S., collaborators: Metalib – the penn locally nameless metatheory library. <https://github.com/plclub/metalib> (2008)
- [23] Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electronic Notes in Theoretical Computer Science 171(4), 73 – 93 (2007), proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006)
- [24] Zalakian, U.: Type-checking session-typed π -calculus with Coq. Master’s thesis, University of Glasgow (2019)

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

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



Games and Automata



Solving Mean-Payoff Games via Quasi Dominions^{*}

Massimo Benerecetti , Daniele Dell’Erba , and Fabio Mogavero 

Università degli Studi di Napoli Federico II, Naples, Italy

Abstract. We propose a novel algorithm for the solution of *mean-payoff games* that merges together two seemingly unrelated concepts introduced in the context of parity games, *small progress measures* and *quasi dominions*. We show that the integration of the two notions can be highly beneficial and significantly speeds up convergence to the problem solution. Experiments show that the resulting algorithm performs orders of magnitude better than the asymptotically-best solution algorithm currently known, without sacrificing on the worst-case complexity.

1 Introduction

In this article we consider the problem of solving *mean-payoff games*, namely infinite-duration perfect-information two-player games played on weighted directed graphs, each of whose vertexes is controlled by one of the two players. The game starts at an arbitrary vertex and, during its evolution, each player can take moves at the vertexes it controls, by choosing one of the outgoing edges. The moves selected by the two players induce an infinite sequence of vertexes, called play. The payoff of any prefix of a play is the sum of the weights of its edges. A play is winning if it satisfies the game objective, called *mean-payoff objective*, which requires that the limit of the *mean payoff*, taken over the prefixes lengths, never falls below a given *threshold* ν .

Mean-payoff games have been first introduced and studied by Ehrenfeucht and Mycielski in [20], who showed that positional strategies suffice to obtain the optimal value. A slightly generalized version was also considered by Gurvich *et al.* in [24]. Positional determinacy entails that the decision problem for these games lies in $\text{NPTIME} \cap \text{CoNPTIME}$ [34], and it was later shown to belong to $\text{UPTIME} \cap \text{CoUPTIME}$ [25], being UPTIME the class of unambiguous non-deterministic polynomial time. This result gives the problem a rather peculiar complexity status, shared by very few other problems, such as integer factorization [22], [1] and parity games [25]. Despite various attempts [7, 19, 24, 30, 34], no polynomial-time algorithm for the mean-payoff game problems is known so far.

A different formulation of the game objective allows to define another class of quantitative games, known as *energy games*. The *energy objective* requires that, given an initial value c , called *credit*, the sum of c and the *payoff* of every prefix

^{*} Partially supported by GNCS 2019 & 2020 projects “Metodi Formali per Tecniche di Verifica Combinata” and “Ragionamento Strategico e Sintesi Automatica di Sistemi Multi-Agente”.

of the play never falls below 0. These games, however, are tightly connected to mean-payoff games, as the two type of games have been proved to be log-space equivalent [11]. They are also related to other more complex forms of quantitative games. In particular, unambiguous polynomial-time reductions [25] exist from these games to *discounted payoff* [34] and *simple stochastic games* [18].

Recently, a fair amount of work in formal verification has been directed to consider, besides correctness properties of computational systems, also quantitative specifications, in order to express performance measures and resource requirements, such as quality of service, bandwidth and power consumption and, more generally, bounded resources. Mean-payoff and energy games also have important practical applications in system verification and synthesis. In [14] the authors show how quantitative aspects, interpreted as penalties and rewards associated to the system choices, allow for expressing optimality requirements encoded as mean-payoff objectives for the automatic synthesis of systems that also satisfy parity objectives. With similar application contexts in mind, [9] and [8] further contribute to that effort, by providing complexity results and practical solutions for the verification and automatic synthesis of reactive systems from quantitative specifications expressed in linear time temporal logic extended with mean-payoff and energy objectives. Further applications to temporal networks have been studied in [16] and [15]. Consequently, efficient algorithms to solve mean-payoff games become essential ingredients to tackle these problems in practice.

Several algorithms have been devised in the past for the solution of the decision problem for mean-payoff games, which asks whether there exists a strategy for one of the players that grants the mean-payoff objective. The very first deterministic algorithm was proposed in [34], where it is shown that the problem can be solved with $O(n^3 \cdot m \cdot W)$ arithmetic operations, with n and m the number of positions and moves, respectively, and W the maximal absolute weight in the game. A strategy improvement approach, based on iteratively adjusting a randomly chosen initial strategy for one player until a winning strategy is obtained, is presented in [31], which has an exponential upper bound. The algorithm by Lifshits and Pavlov [29], which runs in time $O(n \cdot m \cdot 2^n \cdot \log_2 W)$, computes the “potential” of each game position, which corresponds to the initial credit that the player needs in order to win the game from that position. Algorithms based on the solution of linear feasibility problems over the tropical semiring have been also provided in [2–4]. The best known deterministic algorithm to date, which requires $O(n \cdot m \cdot W)$ arithmetic operations, was proposed by Brim *et al.* [13]. They adapt to energy and mean-payoff games the notion of progress measures [28], as applied to parity games in [26]. The approach was further developed in [17] to obtain the same complexity bound for the optimal strategy synthesis problem. A strategy-improvement refinement of this technique has been introduced in [12]. Finally, Bjork *et al.* [6] proposed a randomized strategy-improvement based algorithm running in time $\min\{O(n^2 \cdot m \cdot W), 2^{O(\sqrt{n} \cdot \log n)}\}$.

Our contribution is a novel mean-payoff progress measure approach that enriches such measures with the notion of *quasi dominions*, originally introduced in [5] for parity games. These are sets of positions with the property that as

long as the opponent chooses to play to remain in the set, it loses the game for sure, hence its best choice is always to try to escape. A quasi dominion from where escaping is not possible is a winning set for the other player. Progress measure approaches, such as the one of [13], typically focus on finding the best choices of the opponent and little information is gathered on the other player. In this sense, they are intrinsically asymmetric. Enriching the approach with quasi dominions can be viewed as a way to also encode the best choices of the player, information that can be exploited to speed up convergence significantly. The main difficulty here is that suitable lift operators in the new setting do not enjoy monotonicity. Such a property makes proving completeness of classic progress measure approaches almost straightforward, as monotonic operators do admit a least fixpoint. Instead, the lift operator we propose is only inflationary (specifically, non-decreasing) and, while still admitting fixpoints [10, 33], need not have a least one. Hence, providing a complete solution algorithm proves more challenging. The advantages, however, are significant. On the one hand, the new algorithm still enjoys the same worst-case complexity of the best known algorithm for the problem proposed in [13]. On the other hand, we show that there exist families of games on which the classic approach requires a number of operations that can be made arbitrarily larger than the one required by the new approach. Experimental results also witness the fact that this phenomenon is by no means isolated, as the new algorithm performs orders of magnitude better than the algorithm developed in [13].

2 Mean-Payoff Games

A two-player turn-based *arena* is a tuple $\mathcal{A} = \langle \text{Ps}_\oplus, \text{Ps}_\ominus, Mv \rangle$, with $\text{Ps}_\oplus \cap \text{Ps}_\ominus = \emptyset$ and $\text{Ps} \triangleq \text{Ps}_\oplus \cup \text{Ps}_\ominus$, such that $\langle \text{Ps}, Mv \rangle$ is a finite directed graph without sinks. Ps_\oplus (*resp.*, Ps_\ominus) is the set of positions of player \oplus (*resp.*, \ominus) and $Mv \subseteq \text{Ps} \times \text{Ps}$ is a left-total relation describing all possible moves. A *path* in $V \subseteq \text{Ps}$ is a finite or infinite sequence $\pi \in \text{Pth}(V)$ of positions in V compatible with the move relation, *i.e.*, $(\pi_i, \pi_{i+1}) \in Mv$, for all $i \in [0, |\pi| - 1)$. A positional *strategy* for player $\alpha \in \{\oplus, \ominus\}$ on $V \subseteq \text{Ps}$ is a function $\sigma_\alpha \in \text{Str}_\alpha(V) \subseteq (V \cap \text{Ps}_\alpha) \rightarrow \text{Ps}$, mapping each α -position v in the domain of σ_α to position $\sigma_\alpha(v)$ compatible with the move relation, *i.e.*, $(v, \sigma_\alpha(v)) \in Mv$. With $\text{Str}_\alpha(V)$ we denote the set of all α -strategies on V , while Str_α denotes $\bigcup_{V \subseteq \text{Ps}} \text{Str}_\alpha(V)$. A *play* in $V \subseteq \text{Ps}$ from a position $v \in V$ *w.r.t.* a pair of strategies $(\sigma_\oplus, \sigma_\ominus) \in \text{Str}_\oplus(V) \times \text{Str}_\ominus(V)$, called $((\sigma_\oplus, \sigma_\ominus), v)$ -*play*, is a path $\pi \in \text{Pth}(V)$ such that $\pi_0 = v$ and, for all $i \in [0, |\pi| - 1)$, if $\pi_i \in \text{Ps}_\oplus$ then $\pi_{i+1} = \sigma_\oplus(\pi_i)$ else $\pi_{i+1} = \sigma_\ominus(\pi_i)$. The *play function* $\text{play} : (\text{Str}_\oplus(V) \times \text{Str}_\ominus(V)) \times V \rightarrow \text{Pth}(V)$ returns, for each position $v \in V$ and pair of strategies $(\sigma_\oplus, \sigma_\ominus) \in \text{Str}_\oplus(V) \times \text{Str}_\ominus(V)$, the maximal $((\sigma_\oplus, \sigma_\ominus), v)$ -*play* $\text{play}((\sigma_\oplus, \sigma_\ominus), v)$. If a pair $(\sigma_\oplus, \sigma_\ominus) \in \text{Str}_\oplus(V) \times \text{Str}_\ominus(V)$ induces a finite play starting from position $v \in V$, then $\text{play}((\sigma_\oplus, \sigma_\ominus), v)$ identifies the maximal prefix of that play that is contained in V .

A *mean-payoff game* (MPG for short) is a tuple $\mathfrak{D} = \langle \mathcal{A}, \text{Wg}, \text{wg} \rangle$, where \mathcal{A} is an arena, $\text{Wg} \subset \mathbb{Z}$ is a finite set of integer weights, and $\text{wg} : \text{Ps} \rightarrow \text{Wg}$ is a

weight function assigning a weight to each position. Ps^+ (*resp.*, Ps^-) denotes the set of positive-weight positions (*resp.*, non-positive-weight positions). For convenience, we shall refer to non-positive weights as negative weights. Notice that this definition of MPG is equivalent to the classic formulation in which the weights label the moves, instead. The weight function naturally extends to paths, by setting $\text{wg}(\pi) \triangleq \sum_{i=0}^{|\pi|-1} \text{wg}(\pi_i)$. The goal of player \oplus (*resp.*, \ominus) is to maximize (*resp.*, minimize) $v(\pi) \triangleq \liminf_{i \rightarrow \infty} \frac{1}{i} \cdot \text{wg}(\pi_{\leq i})$, where $\pi_{\leq i}$ is the prefix up to index i . Given a threshold ν , a set of positions $V \subseteq \text{Ps}$ is a \oplus -*dominion*, if there exists a \oplus -strategy $\sigma_{\oplus} \in \text{Str}_{\oplus}(V)$ such that, for all \ominus -strategies $\sigma_{\ominus} \in \text{Str}_{\ominus}(V)$ and positions $v \in V$, the induced play $\pi = \text{play}((\sigma_{\oplus}, \sigma_{\ominus}), v)$ satisfies $v(\pi) > \nu$. The pair of winning regions $(\text{Wn}_{\oplus}, \text{Wn}_{\ominus})$ forms a ν -mean partition. Assuming ν integer, the ν -mean partition problem is equivalent to the 0-mean partition one, as we can subtract ν to the weights of all the positions. As a consequence, the MPG decision problem can be equivalently restated as deciding whether player \oplus (*resp.*, \ominus) has a strategy to enforce $\liminf_{i \rightarrow \infty} \frac{1}{i} \cdot \text{wg}(\pi_{\leq i}) > 0$ (*resp.*, $\liminf_{i \rightarrow \infty} \frac{1}{i} \cdot \text{wg}(\pi_{\leq i}) \leq 0$), for all the resulting plays π .

3 Solving Mean-Payoff Games via Progress Measures

The abstract notion of progress measure [28] has been introduced as a way to encode global properties on paths of a graph by means of simpler local properties of adjacent vertexes. In the context of MPGs, the graph property of interest, called *mean-payoff property*, requires that the mean payoff of every infinite path in the graph be non-positive. More precisely, in game theoretic terms, a *mean-payoff progress measure* witnesses the existence of strategy σ_{\ominus} for player \ominus such that each path in the graph induced by fixing that strategy on the arena satisfies the desired property. A mean-payoff progress measure associates with each vertex of the underlying graph a value, called *measure*, taken from the set of extended natural numbers $\mathbb{N}_{\infty} \triangleq \mathbb{N} \cup \{\infty\}$, endowed with an ordering relation \leq and an addition operation $+$, which extend the standard ordering and addition over the naturals in the usual way. Measures are associated with positions in the game and the measure of a position v can intuitively be interpreted as an estimate of the payoff that player \oplus can enforce on the plays starting in v . In this sense, they measure “how far” v is from satisfying the mean-payoff property, with the maximal measure ∞ denoting failure of the property for v . More precisely, the \ominus -strategy induced by a progress measure ensures that measures do not increase along the paths of the induced graph. This ensures that every path eventually gets trapped in a non-positive-weight cycle, witnessing a win for player \ominus .

To obtain a progress measure, one starts from some suitable association of position of the game with measures. The local information encoded by these measures is then propagated back along the edges of the underlying graph so as to associate with each position the information gathered along plays of some finite length starting from that position. The propagation process is performed according to the following intuition. The measures of positions adjacent to v are propagated back to v only if those measures push v further away from the

property. This propagation is achieved by means of a measure stretch operation $+$, which adds, when appropriate, the measure of an adjacent position to the weight of a given position. This is established by comparing the measure of v with those of its adjacent positions, since, for each position v , the mean-payoff property is defined in terms of the sum of the weights encountered along the plays from that position. The process ends when no position can be pushed further away from the property and each position is not dominated by any, respectively one, of its adjacents, depending on whether that position belongs to player \oplus or to player \ominus , respectively. The positions that did not reach measure ∞ are those from which player \ominus can win the game and the set of measures currently associated with such positions forms a mean-payoff progress measure.

To make the above intuitions precise, we introduce the notion of measure function, progress measure, and an algorithm for computing progress measures correctly. It is worth noticing that the progress-measure based approach as described in [13], called SEPM from now on, can be easily recast equivalently in the form below. A *measure function* $\mu: \text{Ps} \rightarrow \mathbb{N}_\infty$ maps each position v in the game to a suitable measure $\mu(v)$. The order \leq of the measures naturally induces a pointwise partial order \sqsubseteq on the measure functions defined in the usual way, namely, for any two measure functions μ_1 and μ_2 , we write $\eta_1 \sqsubseteq \eta_2$ if $\mu_1(v) \leq \mu_2(v)$, for all positions v . The set of measure functions over a measure space, together with the induced ordering \sqsubseteq , forms a *measure-function space*.

Definition 1 (Measure-Function Space). *The measure-function space is the partial order $\mathcal{F} \triangleq \langle \text{MF}, \sqsubseteq \rangle$ whose components are defined as follows:*

1. $\text{MF} \triangleq \text{Ps} \rightarrow \mathbb{N}_\infty$ is the set of all functions $\mu \in \text{MF}$, called *measure functions*, mapping each position $v \in \text{Ps}$ to a measure $\mu(v) \in \mathbb{N}_\infty$;
2. for all $\mu_1, \mu_2 \in \text{MF}$, it holds that $\mu_1 \sqsubseteq \mu_2$ if $\mu_1(v) \leq \mu_2(v)$, for all $v \in \text{Ps}$.

The \oplus -denotation (resp., \ominus -denotation) of a measure function $\mu \in \text{MF}$ is the set $\|\mu\|_\oplus \triangleq \mu^{-1}(\infty)$ (resp., $\|\mu\|_\ominus \triangleq \mu^{-1}(\infty)$) of all positions having maximal (resp., non-maximal) measure associated within μ .

Consider a position v with an adjacent u with measure η . A measure update of η w.r.t. v is obtained by the stretch operator $+$: $\mathbb{N}_\infty \times \text{Ps} \rightarrow \mathbb{N}_\infty$, defined as $\eta + v \triangleq \max\{0, \eta + \text{wg}(v)\}$, which corresponds to the payoff estimate that the given position will obtain by choosing to follow the move leading to the u .

A *mean-payoff progress measure* is such that the measure associated with each game position v need not be increased further in order to beat the actual payoff of the plays starting from v . In particular, it can be defined by taking into account the opposite attitude of the two players in the game. While the player \oplus tries to push toward higher measures, the player \ominus will try to keep the measures as low as possible. A measure function in which the payoff of each \oplus -position (resp., \ominus -position) v is not dominated by the payoff of all (resp., some of) its adjacents augmented with the weight of v itself meets the requirements.

Definition 2 (Progress Measure). *A measure function $\mu \in \text{MF}$ is a progress measure if the following two conditions hold true, for all positions $v \in \text{Ps}$:*

1. $\mu(u) + v \leq \mu(v)$, for all adjacents $u \in Mv(v)$ of v , if $v \in \text{Ps}_{\oplus}$;
2. $\mu(u) + v \leq \mu(v)$, for some adjacent $u \in Mv(v)$ of v , if $v \in \text{Ps}_{\ominus}$.

The following theorem states the fundamental property of progress measures, namely, that every position with a non-maximal measures is won by player \ominus .

Theorem 1 (Progress Measure). $\|\mu\|_{\ominus} \subseteq \text{Wn}_{\ominus}$, for all progress measures μ .

In order to obtain a progress measure from a given measure function, one can iteratively adjust the current measure values in such a way to force the progress condition above among adjacent positions. To this end, we define the *lift operator* $\text{lift}: \text{MF} \rightarrow \text{MF}$ as follows:

$$\text{lift}(\mu)(v) \triangleq \begin{cases} \max\{\mu(w) + v : w \in Mv(v)\}, & \text{if } v \in \text{Ps}_{\oplus}; \\ \min\{\mu(w) + v : w \in Mv(v)\}, & \text{otherwise.} \end{cases}$$

Note that the lift operator is clearly monotone and, therefore, admits a least fixpoint. A mean-payoff progress measure can be obtained by repeatedly applying this operator until a fixpoint is reached, starting from the minimal measure function $\mu_o \triangleq \{v \in \text{Ps} \mapsto 0\}$ that assigns measure 0 to all the positions in the game. The following *solver operator* applied to μ_o computes the desired solution: $\text{sol} \triangleq \text{lfp } \mu . \text{lift}(\mu): \text{MF} \rightarrow \text{MF}$. Observe that the measures generated by the procedure outlined above have a fairly natural interpretation. Each positive measure, indeed, under-approximates the weight that player \oplus can enforce along finite prefixes of the plays from the corresponding positions. This follows from the fact that, while player \oplus maximizes its measures along the outgoing moves, player \ominus minimizes them. In this sense, each positive measure witnesses the existence of a positively-weighted finite prefix of a play that player \oplus can enforce. Let $S \triangleq \sum\{\text{wg}(v) \in \mathbb{N} : v \in \text{Ps} \wedge \text{wg}(v) > 0\}$ be the sum of all the positive weights in the game. Clearly, the maximal payoff of a simple play in the underlying graph cannot exceed S . Therefore, a measure greater than S witnesses the existence of a cycle whose payoff diverges to infinity and is won, thus, by player \oplus . Hence, any measure strictly greater than S can be substituted with the value ∞ . This observation establishes the termination of the algorithm and is instrumental to its completeness proof. Indeed, at the fixpoint, the measures actually coincide with the highest payoff player \oplus is able to guarantee. Soundness and completeness of the above procedure have been established in [13], where the authors also show that, despite the algorithm requiring $O(n \cdot S) = O(n^2 \cdot W)$ lift operations in the worst-case, with n the number of positions and W the maximal positive weight in the game, the overall cost of these lift operations is $O(S \cdot m \cdot \log S) = O(n \cdot m \cdot W \cdot \log(n \cdot W))$, with m the number of moves and $O(\log S)$ the cost of arithmetic operations to compute the stretch of the measures.

4 Solving Mean-Payoff Games via Quasi Dominions

Let us consider the simple example game depicted in Figure 1, where the shape of each position indicates the owner, circles for player \oplus and square for its

opponent \ominus , and, in each label of the form ℓ/w , the letter w corresponds to the associated weight, where we assume $k > 1$. Starting from the smallest measure function $\mu_0 = \{a, b, c, d \mapsto 0\}$, the first application of the lift operator returns $\mu_1 = \{a \mapsto k; b, c \mapsto 0; d \mapsto 1\} = \text{lift}(\mu_0)$. After that step, the following iterations of the fixpoint alternatively updates positions c and d , since the other ones already satisfy the progress condition. Being $c \in \text{Ps}_{\ominus}$, the lift operator chooses for it the measure computed along the move (c, d) , thus obtaining $\mu_2(c) = \text{lift}(\mu_1)(c) = \mu_1(d) = 1$. Subsequently, d is updated to $\mu_3(d) = \text{lift}(\mu_2)(d) = \mu_2(c) + 1 = 2$. A progress measure is obtained after exactly $2k + 1$ iterations, when the measure of c reaches value k and d value $k + 1$. Note, however, that the choice of the move (c, d) is clearly a losing strategy for player \ominus , as remaining in the highlighted region would make the payoff from position c diverge. Therefore, the only reasonable choice for player \ominus is to exit from that region by taking the move leading to position a . An operator able to diagnose this phenomenon early on could immediately discard the move (c, d) and jump directly to the correct payoff obtained by choosing the move to position a . As we shall see, such an operator might lose the monotonicity property and recovering the completeness of the resulting approach will prove more involved.

In the rest of this article we devise a progress operator that does precisely that. We start by providing a notion of *quasi dominion*, originally introduced for parity games in [5], which can be exploited in the context of MPGs.

Definition 3 (Quasi Dominion). *An set of positions $Q \subseteq \text{Ps}$ is a quasi \oplus -dominion if there exists a \oplus -strategy $\sigma_{\oplus} \in \text{Str}_{\oplus}(Q)$, called \oplus -witness for Q , such that, for all \ominus -strategies $\sigma_{\ominus} \in \text{Str}_{\ominus}(Q)$ and positions $v \in Q$, the play $\pi = \text{play}((\sigma_{\oplus}, \sigma_{\ominus}), v)$, called (σ_{\oplus}, v) -play in Q , satisfies $\text{wg}(\pi) > 0$. If the condition $\text{wg}(\pi) > 0$ holds only for infinite plays π , then Q is called weak quasi \oplus -dominion.*

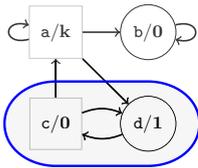


Fig. 1: An MPG.

Essentially, a quasi \oplus -dominion consists in a set Q of positions starting from which player \oplus can force plays in Q of positive weight. Analogously, any infinite play that player \oplus can force in a weak quasi \oplus -dominion has positive weight. Clearly, any quasi \oplus -dominion is also a weak quasi \oplus -dominion. Moreover, the latter are closed under subsets, while the former are

not. It is an immediate consequence of the definition above that all infinite plays induced by the \oplus -witness, if any, necessarily have infinite weight and, thus, are winning for player \oplus . Indeed, every such a play π is regular, *i.e.* it can be decomposed into a prefix π' and a simple cycle $(\pi'')^\omega$, *i.e.* $\pi = \pi'(\pi'')^\omega$, since the strategies we are considering are memoryless. Now, $\text{wg}((\pi'')^\omega) > 0$, so, $\text{wg}(\pi'') > 0$, which implies $\text{wg}((\pi'')^\omega) = \infty$. Hence, $\text{wg}(\pi) = \infty$.

Proposition 1. *Let Q be a weak quasi \oplus -dominion with $\sigma_{\oplus} \in \text{Str}_{\oplus}(Q)$ one of its \oplus -witnesses and $Q^* \subseteq Q$. Then, for all \ominus -strategies $\sigma_{\ominus} \in \text{Str}_{\ominus}(Q^*)$ and positions $v \in Q^*$ the following holds: if the $(\sigma_{\oplus}|_{Q^*}, v)$ -play $\pi = \text{play}((\sigma_{\oplus}|_{Q^*}, \sigma_{\ominus}), v)$ is infinite, then $\text{wg}(\pi) = \infty$.*

From Proposition 1, it directly follows that, if a weak quasi \oplus -dominion Q is closed w.r.t. its \oplus -witness, namely all the induced plays are infinite, then it is a \oplus -dominion, hence is contained in Wn_{\oplus} .

Consider again the example of Figure 1. The set of position $Q \triangleq \{a, c, d\}$ forms a quasi \oplus -dominion whose \oplus -witness is the only possible \oplus -strategy mapping position d to c . Indeed, any infinite play remaining in Q forever and compatible with that strategy (e.g., the play from position c when player \ominus chooses the move from c leading to d or the one from a to itself or the one from a to d) grants an infinite payoff. Any finite compatible play, instead, ends in position a (e.g., the play from c when player \ominus chooses the move from c to a and then one from a to b) giving a payoff of at least $k > 0$. On the other hand, $Q^* \triangleq \{c, d\}$ is only a weak quasi \oplus -dominion, as player \ominus can force a play of weight 0 from position c , by choosing the exiting move (c, a) . However, the internal move (c, d) would lead to an infinite play in Q^* of infinite weight.

The crucial observation here is that the best choice for player \ominus in any position of a (weak) quasi \oplus -dominion is to exit from it as soon as it can, while the best choice for player \oplus is to remain inside it as long as possible. The idea of the algorithm we propose in this section is to precisely exploit the information provided by the quasi dominions in the following way. Consider the example above. In position a player \ominus must choose to exit from $Q = \{a, c, d\}$, by taking the move (a, b) , without changing its measure, which would corresponds to its weight k . On the other hand, the best choice for player \ominus in position c is to exit from the weak quasi-dominion $Q^* = \{c, d\}$, by choosing the move (c, a) and lifting its measure from 0 to k . Note that this contrasts with the minimal measure-increase policy for player \ominus employed in [13], which would keep choosing to leave c in the quasi-dominion by following the move to d , which gives the minimal increase in measure of value 1. Once c is out of the quasi-dominion, though, the only possible move for player \oplus is to follow c , taking measure $k + 1$. The resulting measure function is the desired progress measure.

In order to make this intuitive idea precise, we need to be able to identify quasi dominions first. Interestingly enough, the measure functions μ defined in the previous section do allow to identify a quasi dominion, namely the set of positions $\mu^{-1}(0)$ having positive measure. Indeed, as observed at the end of that section, a positive measure witnesses the existence of a positively-weighted finite play that player \oplus can enforce from that position onward, which is precisely the requirement of Definition 3. In the example of Figure 1, $\mu_0^{-1}(0) = \emptyset$ and $\mu_1^{-1}(0) = \{a, c, d\}$ are both quasi dominions, the first one w.r.t. the empty \oplus -witness and the second one w.r.t. the \oplus -witness $\sigma_{\oplus}(d) = c$.

We shall keep the quasi-dominion information in pairs (μ, σ) , called *quasi-dominion representations* (QDR, for short), composed of a measure function μ and a \oplus -strategy σ , which corresponds to one of the \oplus -witnesses of the set of positions with positive measure in μ . The connection between these two components is formalized in the definition below that also provides the partial order over which the new algorithm operates.

Definition 4 (QDR Space). *The quasi-dominion-representation space is the partial order $\mathcal{Q} \triangleq \langle \text{QDR}, \sqsubseteq \rangle$, whose components are defined as follows:*

1. $\text{QDR} \subseteq \text{MF} \times \text{Str}_{\oplus}$ is the set of all pairs $\varrho \triangleq (\mu_{\varrho}, \sigma_{\varrho}) \in \text{QDR}$, called quasi-dominion-representations, composed of a measure function $\mu_{\varrho} \in \text{MF}$ and a \oplus -strategy $\sigma_{\varrho} \in \text{Str}_{\oplus}(\mathcal{Q}(\varrho))$, where $\mathcal{Q}(\varrho) \triangleq \overline{\mu_{\varrho}^{-1}(0)}$, for which:
 - (a) $\mathcal{Q}(\varrho)$ is a quasi \oplus -dominion enjoying σ_{ϱ} as a \oplus -witness;
 - (b) $\|\mu_{\varrho}\|_{\oplus}$ is a \oplus -dominion;
 - (c) $\mu_{\varrho}(v) \leq \mu_{\varrho}(\sigma_{\varrho}(v)) + v$, for all \oplus -positions $v \in \mathcal{Q}(\varrho) \cap \text{Ps}_{\oplus}$;
 - (d) $\mu_{\varrho}(v) \leq \mu_{\varrho}(u) + v$, for all \ominus -positions $v \in \mathcal{Q}(\varrho) \cap \text{Ps}_{\ominus}$ and $u \in \text{Mv}(v)$;
2. for all $\varrho_1, \varrho_2 \in \text{QDR}$, it holds that $\varrho_1 \sqsubseteq \varrho_2$ if $\mu_{\varrho_1} \sqsubseteq \mu_{\varrho_2}$ and $\sigma_{\varrho_1}(v) = \sigma_{\varrho_2}(v)$, for all \oplus -positions $v \in \mathcal{Q}(\varrho_1) \cap \text{Ps}_{\oplus}$ with $\mu_{\varrho_1}(v) = \mu_{\varrho_2}(v)$.

The α -denotation $\|\varrho\|_{\alpha}$ of a QDR ϱ , with $\alpha \in \{\oplus, \ominus\}$, is the α -denotation $\|\mu_{\varrho}\|_{\alpha}$ of its measure function.

Condition 1a is obvious. Condition 1b, instead, requires that every position with infinite measure is indeed won by player \oplus and is crucial to guarantee the completeness of the algorithm. Finally, Conditions 1c and 1d ensure that every positive measure under approximates the actual weight of some finite play within the induced quasi dominion. This is formally captured by the following proposition, which can be easily proved by induction on the length of the play.

Proposition 2. *Let ϱ be a QDR and $v\pi u$ a finite path starting at position $v \in \text{Ps}$ and terminating in position $u \in \text{Ps}$ compatible with the \oplus -strategy σ_{ϱ} . Then, $\mu_{\varrho}(v) \leq \text{wg}(v\pi) + \mu_{\varrho}(u)$.*

It is immediate to see that every MPG admits a non-trivial QDR space, since the pair (μ_o, σ_o) , with μ_o the smallest measure function and σ_o the empty strategy, trivially satisfies all the required conditions.

Proposition 3. *Every MPG has a non-empty QDR space associated with it.*

The solution procedure we propose, called QDPM from now on, can intuitively be broken down as an alternation of two phases. The first one tries to lift the measures of positions outside the quasi dominion $\mathcal{Q}(\varrho)$ in order to extend it, while the second one lifts the positions inside $\mathcal{Q}(\varrho)$ that can be forced to exit from it by player \ominus . The algorithm terminates when no new position can be absorbed within the quasi dominion and no measure needs to be lifted to allow the \ominus -winning positions to exit from it, when possible. To this end, we define a controlled lift operator $\text{lift}: \text{QDR} \times 2^{\text{Ps}} \times 2^{\text{Ps}} \rightarrow \text{QDR}$ that works on QDRs and takes two additional parameters, a source and a target set of positions. The intended meaning is that we want to restrict the application of the lift operation to the positions in the source set S , while using only the moves leading to the target set T . The different nature of the two types of lifting operations is reflected in the actual values of the source and target parameters.

$$\text{lift}(\varrho, S, T) \triangleq \varrho^*, \text{ where}$$

$$\mu_{\varrho^*}(v) \triangleq \begin{cases} \max\{\mu_{\varrho}(u) + v : u \in Mv(v) \cap T\}, & \text{if } v \in S \cap Ps_{\oplus}; \\ \min\{\mu_{\varrho}(u) + v : u \in Mv(v) \cap T\}, & \text{if } v \in S \cap Ps_{\ominus}; \\ \mu_{\varrho}(v), & \text{otherwise;} \end{cases}$$

and, for all \oplus -positions $v \in Q(\varrho^*) \cap Ps_{\oplus}$, we choose $\sigma_{\varrho^*}(v) \in \operatorname{argmax}_{u \in Mv(v) \cap T} \mu_{\varrho}(u) + v$, if $\mu_{\varrho^*}(v) \neq \mu_{\varrho}(v)$, and $\sigma_{\varrho^*}(v) = \sigma_{\varrho}(v)$, otherwise. Except for the restriction on the outgoing moves considered, which are those leading to the targets in T , the lift operator acts on the measure component of a QDR very much like the original lift operator does. In order to ensure that the result is still a QDR, however, the lift operator must also update the \oplus -witness of the quasi dominion. This is required to guarantee that Conditions 1a and 1c of Definition 4 are preserved. If the measure of a \oplus -position v is not affected by the lift, the \oplus -witness must not change for that position. However, if the application of the lift operation increases the measure, then the \oplus -witness on v needs to be updated to any move (v, u) that grants measure $\mu_{\varrho^*}(v)$ to v . In principle, more than one such move may exist and any one of them can serve as witness.

The solution corresponds to the inflationary fixpoint [10, 33] of the two phases mentioned above, $\operatorname{sol} \triangleq \operatorname{ifp} \varrho. \operatorname{prg}_+(\operatorname{prg}_o(\varrho)) : \text{QDR} \rightarrow \text{QDR}$, defined by the progress operators prg_o and prg_+ . The first phase is computed by the operator $\operatorname{prg}_o : \text{QDR} \rightarrow \text{QDR}$ as follows: $\operatorname{prg}_o(\varrho) \triangleq \operatorname{sup}\{\varrho, \operatorname{lift}(\varrho, \overline{Q(\varrho)}, Ps)\}$. This operator is responsible of enforcing the progress condition on the positions outside the quasi dominion $Q(\varrho)$ that do not satisfy the inequalities between the measures along a move leading to $Q(\varrho)$ itself. It does that by applying the lift operator with $\overline{Q(\varrho)}$ as source and no restrictions on the moves. Those position that acquire a positive measure in this phase contribute to enlarging the current quasi dominion. Observe that the strategy component of the QDR is updated so that it is a \oplus -witness of the new quasi dominion. To guarantee that measures never decrease, the supremum *w.r.t.* the QDR-space ordering is taken as result.

Lemma 1. *μ_{ϱ} is a progress measure over $\overline{Q(\varrho)}$, for all fixpoints ϱ of prg_o .*

The second phase, instead, implements the mechanism intuitively described above, while analyzing the simple example of Figure 1. This is achieved by the operator prg_+ reported in Algorithm 1. The procedure iteratively examines the current quasi dominion and lifts the measures of the positions that must exit from it. Specifically, it processes $Q(\varrho)$ layer by layer, starting from the outer layer of positions that must escape. The process ends when a, possibly empty, closed weak quasi dominion is obtained. Recall that all the positions in a closed weak quasi dominion are necessarily winning for player \oplus , due to Proposition 1. We distinguish two sets of positions in $Q(\varrho)$. Those that already satisfy the progress condition and those that do not. The measures of first ones already witness an escape route from $Q(\varrho)$. The other ones, instead, are those whose current choice is to remain inside it. For instance, when considering the measure function μ_2 in the example of Figure 1, position a belongs to the first set, while positions c and d to the second one, since the choice of c is to follow the internal move (c, d).

Since the only positions that change measure are those in the second set, only such positions need to be examined. To identify them, which form a weak quasi

dominion $\Delta(\varrho)$ strictly contained in $Q(\varrho)$, we proceed as follows. First, we collect the set $\text{npp}(\varrho)$ of positions in $Q(\varrho)$ that do not satisfy the progress condition, called the *non-progress positions*. Then, we compute the set of positions that will have no choice other than reaching $\text{npp}(\varrho)$, by computing the inflationary fixpoint of a suitable pre operator.

$$\begin{aligned} \text{npp}(\varrho) &\triangleq \{v \in Q(\varrho) \cap \text{Ps}_{\oplus} : \exists u \in Mv(v) . \mu_{\varrho}(v) < \mu_{\varrho}(u) + v\} \\ &\quad \cup \{v \in Q(\varrho) \cap \text{Ps}_{\ominus} : \forall u \in Mv(v) . \mu_{\varrho}(v) < \mu_{\varrho}(u) + v\}. \\ \text{pre}(\varrho, Q) &\triangleq Q \cup \{v \in Q(\varrho) \cap \text{Ps}_{\oplus} : \sigma_{\varrho}(v) \in Q\} \\ &\quad \cup \{v \in Q(\varrho) \cap \text{Ps}_{\ominus} : \forall u \in Mv(v) \setminus Q . \mu_{\varrho}(v) < \mu_{\varrho}(u) + v\}. \end{aligned}$$

The final result is $\Delta(\varrho) \triangleq (\text{ifp } Q . \text{pre}(\varrho, Q))(\text{npp}(\varrho))$. Intuitively, $\Delta(\varrho)$ contains all the \oplus -positions that are forced to reach $\text{npp}(\varrho)$ via the quasi-dominion \oplus -witness and all the \ominus -positions that can only avoid reaching $\text{npp}(\varrho)$ by strictly increasing their measure, which player \ominus wants obviously to prevent.

It is important to observe that, from a functional view-point, the progress operator prg_+ would work just as well if applied to the entire quasi dominion $Q(\varrho)$, since it would simply leave unchanged the measure of those positions that already satisfy the progress condition. However, it is crucial that only the positions in $\Delta(\varrho)$ are processed in order to achieve the best asymptotic complexity bound known to date. We shall reiterate on this point later on.

At each iteration of the while-loop of Algorithm 1, let Q denote the current (weak) quasi dominion, initially set to $\Delta(\varrho)$ (Line 1). It first identifies the positions in Q that can immediately escape from it (Line 2). Those are (i) all the \ominus -position with a move leading outside of Q and (ii) the \oplus -positions v whose \oplus -witness σ_{ϱ} forces v to exit from Q , namely $\sigma_{\varrho}(v) \notin Q$, and that cannot strictly increase their measure by choosing to remain in Q . While the condition

Alg. 1: Progress Operator

signature $\text{prg}_+ : \text{QDR} \rightarrow \text{QDR}$
function $\text{prg}_+(\varrho)$

```

1  Q ← Δ(ρ)
2  while esc(ρ, Q) ≠ ∅ do
3    E ← bep(ρ, Q)
4    ρ ← lift(ρ, E, Q̄)
5    Q ← Q \ E
6  ρ ← win(ρ, Q)
7  return ρ

```

for \ominus -position is obvious, the one for \oplus -positions require some explanation. The crucial observation here is that, while player \oplus does indeed prefer to remain in the quasi dominion, it can only do so while ensuring that by changing strategy it does not enable infinite plays within Q that are winning for the adversary. In other words, the new \oplus -strategy must still be a \oplus -witness for Q and this can only be ensured if the new choice strictly increases its measure. The operator $\text{esc} : \text{QDR} \times 2^{\text{Ps}} \rightarrow 2^{\text{Ps}}$ formalizes the idea:

$$\begin{aligned} \text{esc}(\varrho, Q) &\triangleq \{v \in Q \cap \text{Ps}_{\ominus} : Mv(v) \setminus Q \neq \emptyset\} \\ &\quad \cup \{v \in Q \cap \text{Ps}_{\oplus} : \sigma_{\varrho}(v) \notin Q \wedge \forall u \in Mv(v) \cap Q . \mu_{\varrho}(u) + v \leq \mu_{\varrho}(v)\}. \end{aligned}$$

Consider, for instance, the example in Figure 2 and a QDR ϱ such that $\mu_{\varrho} = \{\mathbf{a} \mapsto 3; \mathbf{b} \mapsto 2; \mathbf{c}, \mathbf{d}, \mathbf{f} \mapsto 1; \mathbf{e} \mapsto 0\}$ and $\sigma_{\varrho} = \{\mathbf{b} \mapsto \mathbf{a}; \mathbf{f} \mapsto \mathbf{d}\}$. In this case,

we have $Q_\varrho = \{a, b, c, d, f\}$ and $\Delta(\varrho) = \{c, d, f\}$, since c is the only non-progress positions, d is forced to follow c in order to avoid the measure increase required to reach b , and f is forced by the \oplus -witness to reach d . Now, consider the situation where the current weak quasi dominion is $Q = \{c, f\}$, *i.e.* after d has escaped from $\Delta(\varrho)$. The escape set of Q is $\{c, f\}$. To see why the \oplus -position f is escaping, observe that $\mu_\varrho(f) + f = 1 = \mu_\varrho(f)$ and that, indeed, should player \oplus choose to change its strategy and take the move (f, f) to remain in Q , it would obtain an infinite play with payoff 0, thus violating the definition of weak quasi dominion.

Before proceeding, we want to stress an easy consequence of the definition of the notion of escape set and Conditions 1c and 1d of Definition 4, *i.e.*, that every escape position of the quasi dominion $Q(\varrho)$ can only assume its weight as possible measure inside a QDR ϱ , as reported is the following proposition. This observation, together with Proposition 2, ensures that the measure of a position $v \in Q(\varrho)$ is an under approximation of the weight of all finite plays leaving $Q(\varrho)$.

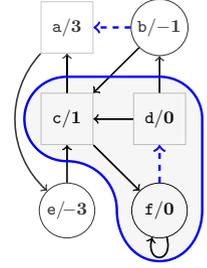


Fig. 2: Another MPG.

Proposition 4. *Let ϱ be a QDR. Then, $\mu_\varrho(v) = \text{wg}(v) > 0$, for all $v \in \text{esc}(\varrho, Q(\varrho))$.*

Now, going back to the analysis of the algorithm, if the escape set is non-empty, we need to select the escape positions that need to be lifted in order to satisfy the progress condition. The main difficulty is to do so in such a way that the resulting measure function still satisfies Condition 1d of Definition 4, for all the \boxminus -positions with positive measure. The problem occurs when a \boxminus -position can exit either immediately or passing through a path leading to another position in the escape set. Consider again the example above, where $Q = \Delta(\varrho) = \{c, d, f\}$. If position d immediately escapes from Q using the move (d, b) , it would change its measure to $\mu'(d) = \mu(b) + d = 2 > \mu(d) = 1$. Now, position c has two ways to escape, either directly with move (c, a) or by reaching the other escape position d passing through f . The first choice would set its measure to $\mu(a) + c = 4$. The resulting measure function, however, would not satisfy Condition 1d of Definition 4, as the new measure of c would be greater than $\mu'(d) + c = 2$, preventing to obtain a QDR. Similarly, if position d escapes from Q passing through c via the move (c, a) , we would have $\mu''(d) = \mu''(c) + d = (\mu(a) + c) + d = 4 > 2 = \mu(b) + d$, still violating Condition 1d. Therefore, in this specific case, the only possible way to escape is to reach b . The solution to this problem is simply to lift in the current iteration only those positions that obtain the lowest possible measure increase, hence position d in the example, leaving the lift of c to some subsequent iteration of the algorithm that would choose the correct escape route via d . To do so, we first compute the minimal measure increase, called the *best-escape forfeit*, that each position in the escape set would obtain by exiting the quasi dominion immediately. The positions with the lowest possible forfeit, called *best-escape positions*, can all be lifted at the same time. The intuition is that the measure of all the positions that escape from a (weak) quasi dominion will necessarily be increased of at least the minimal best-escape forfeit. This observation is at the core of the proof of Theorem 2

(see the appendix) ensuring that the desired properties of QDRs are preserved by the operator prg_+ . The set of best-escape positions is computed by the operator $\text{bep}: \text{QDR} \times 2^{\text{Ps}} \rightarrow 2^{\text{Ps}}$ as follows: $\text{bep}(\varrho, \mathcal{Q}) \triangleq \text{argmin}_{v \in \text{esc}(\varrho, \mathcal{Q})} \text{bef}(\mu_\varrho, \mathcal{Q}, v)$, where the operator $\text{bef}: \text{MF} \times 2^{\text{Ps}} \times \text{Ps} \rightarrow \mathbb{N}_\infty$ computes, for each position v in a quasi dominion \mathcal{Q} , its best-escape forfeit:

$$\text{bef}(\mu, \mathcal{Q}, v) \triangleq \begin{cases} \max\{\mu(u) + v - \mu(v) : u \in Mv(v) \setminus \mathcal{Q}\}, & \text{if } v \in \text{Ps}_\oplus; \\ \min\{\mu(u) + v - \mu(v) : u \in Mv(v) \setminus \mathcal{Q}\}, & \text{otherwise.} \end{cases}$$

In our example, $\text{bef}(\mu, \mathcal{Q}, c) = \mu(a) + c - \mu(c) = 4 - 1 = 3$, while $\text{bef}(\mu, \mathcal{Q}, d) = \mu(b) + d - \mu(d) = 2 - 1 = 1$. Therefore, $\text{bep}(\varrho, \mathcal{Q}) = \{d\}$.

Once the set E of best-escape positions is identified (Line 3), the procedure lifts them restricting the possible moves to those leading outside the current quasi dominion (Line 4). Those positions are, then, removed from the set (Line 5), thus obtaining a smaller weak quasi dominion ready for the next iteration.

The algorithm terminates when the (possibly empty) current quasi dominion \mathcal{Q} is closed. By virtue of Proposition 1, all those positions belong to Wn_\oplus and their measure is set to ∞ by means of the operator $\text{win}: \text{QDR} \times 2^{\text{Ps}} \rightarrow \text{QDR}$ (Line 6), which also computes the winning \oplus -strategy on those positions, as follows: $\text{win}(\varrho, \mathcal{Q}) \triangleq \varrho^*$, where $\mu_{\varrho^*} \triangleq \mu_\varrho[\mathcal{Q} \mapsto \infty]$ and, for all \oplus -positions $v \in \mathcal{Q}(\varrho^*) \cap \text{Ps}_\oplus$, we choose $\sigma_{\varrho^*}(v) \in \text{argmax}_{u \in Mv(v) \cap \mathcal{Q}} \mu_\varrho(u) + v$, if $\sigma_\varrho(v) \notin \mathcal{Q}$, and $\sigma_{\varrho^*}(v) = \sigma_\varrho(v)$, otherwise. Observe that, since we know that every \oplus -position $v \in \mathcal{Q} \cap \text{Ps}_\oplus$, whose current \oplus -witness leads outside \mathcal{Q} , is not an escape position, any move (v, u) within \mathcal{Q} that grants the maximal stretch $\mu_\varrho(u) + v$ strictly increases its measure and, therefore, is a possible choice for a \oplus -witness of the \oplus -dominion \mathcal{Q} .

At this point, it should be quite evident that the progress operator prg_+ is responsible of enforcing the progress condition on the positions inside the quasi dominion $\mathcal{Q}(\varrho)$, thus, the following necessarily holds.

Lemma 2. μ_ϱ is a progress measure over $\mathcal{Q}(\varrho)$, for all fixpoints ϱ of prg_+ .

In order to prove the correctness of the proposed algorithm, we first need to ensure that any quasi-dominion space \mathcal{Q} is indeed closed under the operators prg_o and prg_+ . This is established by the following theorem, which states that the operators are total functions on that space.

Theorem 2. The operators prg_o and prg_+ are total inflationary functions.

Since both operators are inflationary, so is their composition, which admits fixpoint. Therefore, the operator sol is well defined. Moreover, following the same considerations discussed at the end of Section 3, it can be proved the fixpoint is obtained after at most $n \cdot (S + 1)$ iterations. Let $\text{ifp}_k X.F(X)$ denote the k -th iteration of an inflationary operator F . Then, we have the following theorem.

Theorem 3 (Termination). The solver operator $\text{sol} \triangleq \text{ifp } \varrho. \text{prg}_+(\text{prg}_o(\varrho))$ is a well-defined total function. Moreover, for every $\varrho \in \text{QDR}$ it holds that $\text{sol}(\varrho) = (\text{ifp}_k \varrho^* \cdot \text{prg}_+(\text{prg}_o(\varrho^*))) (\varrho)$, for some index $k \leq n \cdot (S + 1)$, where n is the number of positions in the MPG and $S \triangleq \sum \{\text{wg}(v) \in \mathbb{N} : v \in \text{Ps} \wedge \text{wg}(v) > 0\}$ the total sum of its positive weights.

As already observed before, Figure 1 exemplifies an infinite family of games with a fixed number of positions and increasing maximal weight k over which the SEPM algorithm requires $2k + 1$ iterations of the lift operator. On the contrary, QDPM needs exactly two iterations of the solver operator `sol` to find the progress measure, starting from the smallest measure function μ_0 . Indeed, the first iteration returns a measure function $\mu_1 = \text{sol}(\mu_0)$, with $\mu_1(\mathbf{a}) = k$, $\mu_1(\mathbf{b}) = \mu_1(\mathbf{c}) = 0$, and $\mu_1(\mathbf{d}) = 1$, while the second one $\mu_2 = \text{sol}(\mu_1)$ identifies the smallest progress measure, with $\mu_2(\mathbf{a}) = \mu_2(\mathbf{c}) = k$, $\mu_2(\mathbf{b}) = 0$, and $\mu_2(\mathbf{d}) = k + 1$. From this observations, the next result immediately follows.

Theorem 4. *An infinite family of MPGs $\{\mathcal{D}_k\}_k$ exists on which QDPM requires a constant number of measure updates, while SEPM requires $O(k)$ such updates.*

From Theorem 1 and Lemmas 1 and 2 it follows that the solution provided by the algorithm is indeed a progress measure, hence establishing soundness. Completeness follows from Theorem 3 and from Condition 1b of Definition 4 that ensures that all the positions with infinite measure are winning for player \oplus .

Theorem 5 (Correctness). $\|\text{sol}(\varrho)\|_{\square} = \text{Wn}_{\square}$, for every $\varrho \in \text{QDR}$.

The following lemma ensures that each execution of the operator `prg+` strictly increases the measure of all the positions in $\Delta(\varrho)$.

Lemma 3. *Let $\varrho^* \triangleq \text{prg}_+(\varrho)$. Then, $\mu_{\varrho^*}(v) > \mu_{\varrho}(v)$, for all positions $v \in \Delta(\varrho)$.*

Recall that each position can at most be lifted $S + 1 = O(n \cdot W)$ times and, by the previous lemma, the complexity of `sol` only depends on the cumulative cost of such lift operations. We can express, then, the total cost as the sum, over the set of positions in the game, of the cost of all the lift operations performed on that positions. Each such operation can be computed in time linear in the number of incoming and outgoing moves of the corresponding lifted position v , namely $O((|Mv(v)| + |Mv^{-1}(v)|) \cdot \log S)$, with $O(\log S)$ the cost of each arithmetic operation involved. Summing all up, the actual asymptotic complexity of the procedure can, therefore, be expressed as $O(n \cdot m \cdot W \cdot \log(n \cdot W))$.

Theorem 6 (Complexity). *QDPM requires time $O(n \cdot m \cdot W \cdot \log(n \cdot W))$ to solve an MPG with n positions, m moves, and maximal positive weight W .*

5 Experimental Evaluation

In order to assess the effectiveness of the proposed approach, we implemented both QDPM and SEPM [13], the most efficient known solution to the problem and the more closely related one to QDPM, in C++ within OINK [32]. OINK has been developed as a framework to compare parity game solvers. However, extending the framework to deal with MPGs is not difficult. The form of the arenas of the two types of games essentially coincide, the only relevant difference being that MPGs allow negative numbers to label game positions. We ran the two solvers against randomly generated MPGs of various sizes.¹

¹ The experiments were carried out on a 64-bit 3.9GHz quad-core machine, with INTEL i5-6600K processor and 8GB of RAM, running UBUNTU 18.04.

Figure 3 compares the solution time, expressed in seconds, of the two algorithms on 4000 games, each with 10^4 positions and randomly assigned weights in the range $[-15 \times 10^3, 15 \times 10^3]$. The scale of both axes is logarithmic. The experiments are divided in 4 clusters, each containing 1000 games. The benchmarks in different clusters differ in the maximal number m of outgoing moves per position, with $m \in \{10, 20, 40, 80\}$. These experiments clearly show that QDPM substantially outperforms SEPM. Most often, the gap between the two algorithms is between two and three orders of magnitude, as indicated by the dashed diagonal lines. It also shows that SEPM is particularly sensitive to the density of the underlying graph, as its performance degrades significantly as the number of moves increases. The maximal solution time was 21000 sec. for SEPM and 0.017 sec. for QDPM. Figure 4, instead, compares the two algorithms fixing the maximal out-degree of the underlying graphs to 2, in the left-hand picture, and to 40, in the right-hand one, while increasing the number of positions from 10^3 to 10^5 along the x-axis. Each picture displays the performance results on 2800 games. Each point shows the total time to solve 100 randomly generated games with that given number of positions, which increases by 1000 up to size $2 \cdot 10^3$ and by 10000, thereafter. In both pictures the scale is logarithmic. For the experiments in the right-hand picture we had to set a timeout for SEPM to 45 minutes per game, which was hit most of the times on the bigger ones. Once again, the QDPM significantly outperforms SEPM on both kinds of benchmarks, with a gap of more than an order of magnitude on the first ones, and a gap of more than three orders of magnitude on the second ones. The results also confirm that the performance gap grows considerably as the number of moves per position increases.

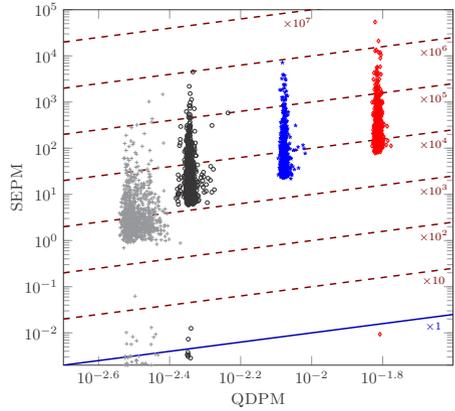


Fig. 3: Random games with 10^4 positions.

Figure 4, instead, compares the two algorithms fixing the maximal out-degree of the underlying graphs to 2, in the left-hand picture, and to 40, in the right-hand one, while increasing the number of positions from 10^3 to 10^5 along the x-axis. Each picture displays the performance results on 2800 games. Each point shows the total time to solve 100 randomly generated games with that given number of positions, which increases by 1000 up to size $2 \cdot 10^3$ and by 10000, thereafter. In both pictures the scale is logarithmic. For the experiments in the right-hand picture we had to set a timeout for SEPM to 45 minutes per game, which was hit most of the times on the bigger ones. Once again, the QDPM significantly outperforms SEPM on both kinds of benchmarks, with a gap of more than an order of magnitude on the first ones, and a gap of more than three orders of magnitude on the second ones. The results also confirm that the performance gap grows considerably as the number of moves per position increases.

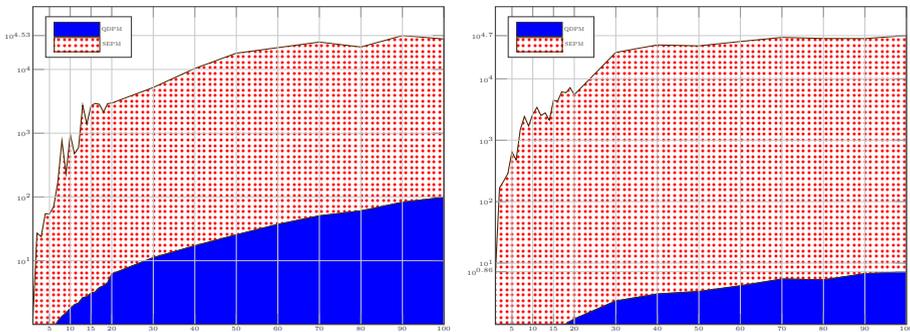


Fig. 4: Total solution times in seconds of SEPM and QDPM on 5600 random games.

We are not aware of actual concrete benchmarks for MPGs. However, exploiting the standard encoding of parity games into mean-payoff games [25], we can compare the behavior of SEPM and QDPM on concrete verification problems encoded as parity games. For completeness, Table 1 reports some experiments on such problems. The table reports the execution times, expressed in seconds, required by the two algorithms to solve instances of two classic verification problems: the Elevator Verification and the Language Inclusion problems. These two benchmarks are included in the PGSolver [23] toolkit and are often used as benchmarks for parity games solvers. The first benchmark is a *verification under fairness* constraints of a simple model of an elevator, while the second one encodes the *language inclusion* problem between a non-deterministic Büchi automaton and a deterministic one. The results on various instances of those problems confirm that QDPM significantly outperforms the classic progress measure approach. Note also that the translation into MPGs, which encodes priorities as weights whose absolute value is exponential in the values of the priorities, leads to games with weights of high magnitude. Hence, the results in Table 1 provide further evidence that QDPM is far less dependent on the absolute value of the weights. They also show that QDPM can be very effective for the solution of real-world qualitative verification problems.

It is worth noting, though, that the translation from parity to MPGs gives rise to weights that are exponentially distant from each other [25]. As a consequence, the resulting benchmarks are not necessarily representative of MPGs, being a very restricted subclass. Nonetheless, they provide evidence of the applicability of the approach in practical scenarios.

Benchmark	Positions	Moves	SEPM	QDPM
Elevator 1	144	234	0.0661	0.0001
Elevator 2	564	950	8.80	0.0003
Elevator 3	2688	4544	4675.71	0.0017
Lang. Incl. 1	170	1094	3.18	0.0021
Lang. Incl. 2	304	1222	16.76	0.0019
Lang. Incl. 3	428	878	20.25	0.0033
Lang. Incl. 4	628	1538	135.51	0.0029
Lang. Incl. 5	509	2126	148.37	0.0034
Lang. Incl. 6	835	2914	834.90	0.0051
Lang. Incl. 7	1658	4544	2277.87	0.0100

Table 1: Concrete verification problems.

6 Concluding Remarks

We proposed a novel solution algorithm for the decision problem of MPGs that integrates progress measures and quasi dominions. We argue that the integration of these two concepts may offer significant speed up in convergence to the solution, at no additional computational cost. This is evidenced by the existence of a family of games on which the combined approach can perform arbitrarily better than a classic progress measure based solution. Experimental results also show that the introduction of quasi dominions can often reduce solution times up to three order of magnitude, suggesting that the approach may be very effective in practical applications as well. We believe that the integration approach we devised is general enough to be applied to other types of games. In particular, the application of quasi dominions in conjunction with progress measure based approaches, such as those of [27] and [21], may lead to practically efficient quasi polynomial algorithms for parity games and their quantitative extensions.

References

1. M. Agrawal, N. Kayal, and N. Saxena, “PRIMES is in P.” *AM*, vol. 160, no. 2, pp. 781–793, 2004.
2. X. Allamigeon, P. Benchimol, and S. Gaubert, “Combinatorial Simplex Algorithms Can Solve Mean-Payoff Games.” *SIAM*, vol. 24, no. 4, pp. 2096–2117, 2014.
3. —, “The Tropical Shadow-Vertex Algorithm Solves Mean-Payoff Games in Polynomial Time on Average.” in *ICALP’14*, 2014, pp. 89–100.
4. X. Allamigeon, P. Benchimol, S. Gaubert, and M. Joswig, “Tropicalizing the Simplex Algorithm.” *SIAM*, vol. 29, no. 2, pp. 751–795, 2015.
5. M. Benerecetti, D. Dell’Erba, and F. Mogavero, “Solving Parity Games via Priority Promotion.” in *CAV’16*, ser. LNCS 9780 (Part II). Springer, 2016, pp. 270–290.
6. H. Björklund, S. Sandberg, and S. Vorobyov, “A Combinatorial Strongly Subexponential Strategy Improvement Algorithm for Mean-Payoff Games.” in *MFCS’04*, 2004, pp. 673–685.
7. H. Björklund and S. Vorobyov, “A Combinatorial Strongly Subexponential Strategy Improvement Algorithm for Mean-Payoff Games.” *DAM*, vol. 155, no. 2, pp. 210–229, 2007.
8. A. Bohy, V. Bruyère, E. Filiot, and J.-F. Raskin, “Synthesis from LTL Specifications with Mean-Payoff Objectives.” in *TACAS’13*, 2013, pp. 169–184.
9. U. Boker, K. Chatterjee, T. Henzinger, and O. Kupferman, “Temporal Specifications with Accumulative Values.” in *LICS’11*, 2011, pp. 43–52.
10. N. Bourbaki, “Sur le Théorème de Zorn.” *AM*, vol. 2, no. 6, pp. 434–437, 1949.
11. P. Bouyer, U. Fahrenberg, K. Larsen, N. Markey, and J. Srba, “Infinite Runs in Weighted Timed Automata with Energy Constraints.” in *FORMATS’2008*. Springer, 2008, pp. 33–47.
12. L. Brim and J. Chaloupka, “Using Strategy Improvement to Stay Alive.” *IJFCS*, vol. 23, no. 3, pp. 585–608, 2012.
13. L. Brim, J. Chaloupka, L. Doyen, R. Gentilini, and J.-F. Raskin, “Faster Algorithms for Mean-Payoff Games.” *FMSD*, vol. 38, no. 2, pp. 97–118, 2011.
14. R. B. K. Chatterjee, T. Henzinger, and B. Jobstmann, “Better Quality in Synthesis Through Quantitative Objectives.” in *CAV’09*, 2009, pp. 140–156.
15. C. Comin, R. Posenato, and R. Rizzi, “Hyper Temporal Networks - A Tractable Generalization of Simple Temporal Networks and its Relation to Mean-Payoff Games.” *Constraints*, vol. 22, no. 2, 2017.
16. C. Comin and R. Rizzi, “Dynamic Consistency of Conditional Simple Temporal Networks via Mean-Payoff Games: A Singly-Exponential Time DC-checking.” in *TIME’15*. IEEECS, 2015, pp. 19–28.
17. —, “Improved Pseudo-Polynomial Bound for the Value Problem and Optimal Strategy Synthesis in Mean-Payoff Games.” *Algorithmica*, vol. 77, no. 4, 2017.
18. A. Condon, “The Complexity of Stochastic Games.” *IC*, vol. 96, no. 2, pp. 203–224, 1992.
19. V. Dhingra and S. Gaubert, “How to Solve Large Scale Deterministic Games with Mean Payoff by Policy Iteration.” in *VALUETOOLS’06*. ACM, 2006, p. 12.
20. A. Ehrenfeucht and J. Mycielski, “Positional Strategies for Mean Payoff Games.” *IJGT*, vol. 8, no. 2, 1979.
21. J. Fearnley, S. Jain, S. Schewe, F. Stephan, and D. Wojtczak, “An Ordered Approach to Solving Parity Games in Quasi Polynomial Time and Quasi Linear Space.” in *SPIN’17*. ACM, 2017, pp. 112–121.

22. M. Fellows and N. Koblitz, “Self-Witnessing Polynomial-Time Complexity and Prime Factorization.” in *CSCT’92*. IEEECS, 1992, pp. 107–110.
23. O. Friedmann and M. Lange, “Solving Parity Games in Practice.” in *ATVA’09*, ser. LNCS 5799. Springer, 2009, pp. 182–196.
24. V. Gurvich, A. Karzanov, and L. Khachivan, “Cyclic Games and an Algorithm to Find Minimax Cycle Means in Directed Graphs.” *USSRCMMP*, vol. 28, no. 5, pp. 85–91, 1988.
25. M. Jurdziński, “Deciding the Winner in Parity Games is in $UP \cap co-UP$.” *IPL*, vol. 68, no. 3, pp. 119–124, 1998.
26. —, “Small Progress Measures for Solving Parity Games.” in *STACS’00*, ser. LNCS 1770. Springer, 2000, pp. 290–301.
27. M. Jurdziński and R. Lazic, “Succinct Progress Measures for Solving Parity Games.” in *LICS’17*. ACM, 2017, pp. 1–9.
28. N. Klarlund, “Progress Measures for Complementation of omega-Automata with Applications to Temporal Logic.” in *FOCS’91*. IEEECS, 1991, pp. 358–367.
29. Y. Lifshits and D. Pavlov, “Potential theory for mean payoff games.” *JMS*, vol. 145, no. 3, pp. 4967–4974, 2007.
30. N. Pisaruk, “Mean-Cost Cyclical Games.” *MOR*, vol. 24, no. 4, pp. 817–828, 1999.
31. S. Schewe, “An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games.” in *CSL’08*, ser. LNCS 5213. Springer, 2008, pp. 369–384.
32. T. van Dijk, “Oink: an Implementation and Evaluation of Modern Parity Game Solvers.” in *TACAS’18*, ser. LNCS 10805. Springer, 2018, pp. 291–308.
33. E. Witt, “Beweisstudien zum Satz von M. Zorn.” *MN*, vol. 4, no. 1-6, pp. 434–438, 1950.
34. U. Zwick and M. Paterson, “The Complexity of Mean Payoff Games on Graphs.” *TCS*, vol. 158, no. 1-2, pp. 343–359, 1996.

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

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





Partial-Order Reduction for Parity Games with an Application on Parameterised Boolean Equation Systems

Thomas Neele^(✉), Tim A.C. Willemse, and Wieger Wesselink

Eindhoven University of Technology, Eindhoven, The Netherlands
{t.s.neele, t.a.c.willemse, j.w.wesselink}@tue.nl

Abstract. Partial-order reduction (POR) is a well-established technique to combat the problem of state-space explosion. We propose POR techniques that are sound for parity games, a well-established formalism for solving a variety of decision problems. As a consequence, we obtain the first POR method that is sound for model checking for the full modal μ -calculus. Our technique is applied to, and implemented for the fixed point logic called *parameterised Boolean equation systems*, which provides a high-level representation of parity games. Experiments indicate that substantial reductions can be achieved.

1 Introduction

In the field of formal methods, model checking [2] is a popular technique to analyse the behaviour of concurrent processes. However, the arbitrary interleaving of these parallel processes can cause an exponential blowup, which is known as the *state-space explosion* problem. Several approaches have been identified to alleviate this issue, by reducing the state space *on-the-fly*, *i.e.*, while generating it. Two established techniques are *symmetry reduction* [13] and *partial-order reduction* (POR) [8,26,30]. Whereas symmetry reduction can only be applied to systems that contain several copies of a component, POR also applies to heterogeneous systems. However, a major drawback of POR is that most variants at best preserve only a fragment of a given logic, such as LTL or CTL* without the next operator (LTL_{-X}/CTL*_{-X}) [7] or the weak modal μ -calculus [28]. Furthermore, the variants of POR that preserve a branching time logic impose significant restrictions on the reduction by only allowing the prioritisation of exactly one action at a time. This decreases the amount of reduction achieved.

In this paper, we address these shortcomings by applying POR on parity games. A parity game is an infinite-duration, two-player game played on a directed graph with decorations on the nodes, in which the players *even* (denoted \diamond) and *odd* (denoted \square) strive to win the nodes of the graph. An application of parity games is encoding a model checking question: a combination of a model, in the form of a transition system, and a formal property, formulated in the modal μ -calculus [16]. In such games, every node v represents the combination

of a state s from the transition system and a (sub)formula φ . Under a typical encoding, player \diamond wins in v if and only if φ holds in s .

In the context of model checking, parity games suffer from the same state-space explosion that models do. Exploring the state space of a parity game under POR can be a very effective way to address this. Our contributions are as follows:

- We propose conditions (Def. 4) that ensure that the *reduction function* used to reduce the parity game is correct, *i.e.*, preserves the winning player of the parity game (Thm. 1).
- We identify improvements for the reduction by investigating the typical structure of a parity game that encodes a model checking question.
- We illustrate how to apply our POR technique in the context of solving *parameterised Boolean equation systems* (PBESs) [10]—a fixed point logic closely related to LFP—as a high-level representation of a parity game.
- We extend the ideas of [17] with support for non-determinism and experiment with an implementation for solving PBESs.

Our approach has two distinct benefits over traditional POR techniques that operate on transition systems. First, it is the first work that enables the use of partial-order reduction for model checking for the full modal μ -calculus. Second, the conditions that we propose are strictly weaker than those necessary to preserve the branching structure of a transition system used in other approaches to POR for branching time logics [7,28], increasing the effectiveness of POR.

The experiments with our implementation for solving PBESs are quite promising. Our results show that, in particular, those instances in which PBESs encode model checking problems involving large state spaces benefit from the use of partial-order reduction. In such cases, a significant size reduction is possible, even when checking complex μ -calculus formulae, and the time penalty of conducting the static analysis is more than made up for by the speed-up in the state space exploration phase.

Related Work There are several proposals for using partial-order reduction for branching-time logics. Groote and Sellink [9] define several forms of *confluence reduction* and prove which behavioural equivalences (and by extension, which fragments of logics) are preserved. In confluence reduction, one tries to identify internal transitions that can safely be prioritised, leading to a smaller state space. Ramakrishna and Smolka [28] propose a notion that coincides with strong confluence from [9], preserving weak bisimilarity and the corresponding logic weak modal μ -calculus.

Similar ideas are presented by Gerth *et al.* in [7]. Their approach is based on the *ample set* method [26] and preserves a relation they call visible bisimulation and the associated logic CTL_{-X} . To preserve the branching structure, they introduce a *singleton proviso* which, contrary to our theory, can greatly impair the amount of reduction that can be achieved (see our Example 3, page 313).

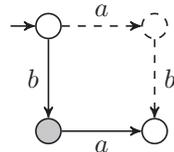
Valmari [33] describes the *stubborn sets* method for LTL_{-X} model checking. In general, stubborn sets allow for larger reductions than ample sets. While investigating the use of stubborn sets for parity games, we identified a subtle issue in one of the stubborn set conditions (called **D1** in [33]). When applied to LTSs

or KSs, this means that $LTL_{\neg X}$ is not necessarily preserved. Moreover, using the condition in the setting of parity games may result in games with different winners; for an example, see our technical report [24]. In [21], we further explore the consequences of the faulty condition for stubborn-set based POR techniques that can be found in the literature. We here resort to a strengthened version of condition **D1** that does not suffer from these issues.

Similar to our approach, Peled [27] applies POR on the product of a transition system and a Büchi automaton, which represents an $LTL_{\neg X}$ property. It is important to note, though, that this original theory is not sound, as discussed in [29]. Kan *et al.* [14] improve on Peled’s ideas and manage to preserve all of LTL. To achieve this, they analyse the Büchi automaton that corresponds to the LTL formula to identify which part is stutter insensitive. With this information, they can reduce the state space in the appropriate places and preserve the validity of the LTL formula under consideration.

The recent work by Bønneland *et al.* [3] is close to ours in spirit, applying stubborn-set based POR to *reachability games*. Such games can be used for synthesis and for model checking reachability properties. Although the conditions on reduction they propose seem unaffected by the aforementioned issue with **D1**, unfortunately, their POR theory is nevertheless unsound, as we next illustrate.

In reachability games, player 1 tries to reach one of the *goal* states, while player 2 tries to avoid them. Bønneland *et al.* propose a condition **R** that guarantees that all goal states in the full game are also reachable in the reduced game. However, the reverse is not guaranteed: paths that do not contain a goal state are not necessarily preserved, essentially endowing player 1 with more power. Consider the (solitaire) reachability game depicted on the right, in which all edges belong to player 2 and the only goal state is indicated with grey. Player 2 wins the non-reduced game by avoiding the goal state via the edges labelled with a and then b . However, $\{b\}$ is a stubborn set—according to the conditions of [3]—in the initial state, and the dashed transitions are thus eliminated in the reduced game. Hence, player 2 is forced to move the token to the goal state and player 1 wins in the reduced game. In the mean time, the authors of [3] confirmed and resolved the issue in [4].



Outline. We give a cursory overview of parity games in Section 2. In Section 3 we introduce partial-order reduction for parity games, and we introduce a further improvement in Section 3.3. Section 4 briefly introduces the PBES fixed point logic, and in Section 5, we describe how to effectively implement parity-game based POR for PBESs. We present the results of our experiments of using parity-game based POR for PBESs in Section 6. We conclude in Section 7.

2 Preliminaries

Parity games are infinite-duration, two-player games played on a directed graph. The objective of the players, called *even* (denoted by \diamond) and *odd* (denoted by \square), is to win nodes in the graph.

Definition 1. A parity game is a directed graph $G = (V, E, \Omega, \mathcal{P})$, where

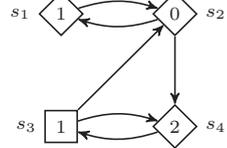
- V is a finite set of nodes, called the state space;
- $E \subseteq V \times V$ is a total edge relation;
- $\Omega : V \rightarrow \mathbb{N}$ is a function that assigns a priority to each node; and
- $\mathcal{P} : V \rightarrow \{\diamond, \square\}$ is a function that assigns a player to each node.

We write $s \rightarrow t$ whenever $(s, t) \in E$. The set of successors of a node s is denoted with $\text{succ}(s) = \{t \mid s \rightarrow t\}$. We use \circ to denote an arbitrary player and $\bar{\circ}$ to denote its opponent.

A parity game is played as follows: initially, a token is placed on some node of the graph. The owner of the node can decide where to move the token; the token may be moved along one of the outgoing edges. This process continues ad infinitum, yielding an infinite path of nodes that the token moves through. Such a path is called a *play*. A play π is won by player \diamond if the minimal priority that occurs infinitely often along π is even. Otherwise, it is won by player \square .

To reason about moves that a player may want to take, we use the concept of *strategies*. A strategy $\sigma_{\circ} : V^+ \rightarrow V$ for player \circ is a partial function that determines where \circ moves the token next, after the token has passed through a finite sequence of nodes. More formally, for all sequences $s_1 \dots s_n$ such that $\mathcal{P}(s_n) = \circ$, it holds that $\sigma_{\circ}(s_1 \dots s_n) \in \text{succ}(s_n)$. If s_n belongs to $\bar{\circ}$, $\sigma_{\circ}(s_1 \dots s_n)$ is undefined. A play s_1, s_2, \dots is *consistent* with a strategy σ if and only if $\sigma(s_1 \dots s_i) = s_{i+1}$ for all i such that $\sigma(s_1 \dots s_i)$ is defined. A player \circ wins in a node s if and only if there is a strategy σ_{\circ} such that all plays that start in s and that are consistent with σ_{\circ} are won by player \circ .

Example 1. Consider the parity game on the right. Here, priorities are inscribed in the nodes and the nodes are shaped according to their owner (\diamond or \square). Let π be an arbitrary, possibly empty, sequence of nodes. In this game, the strategy σ_{\diamond} , partially defined as $\sigma_{\diamond}(\pi s_1) = s_2$ and $\sigma_{\diamond}(\pi s_2) = s_1$, is winning for \diamond in s_1 and s_2 . After all, the minimal priority that occurs infinitely often along $(s_1 s_2)^\omega$ is 0, which is even. Player \square can win node s_3 with the strategy $\sigma_{\square}(\pi s_3) = s_4$. Note that player \diamond is always forced to move the token from node s_4 to s_3 . \square



3 Partial-Order Reduction

In model checking, arbitrary interleaving of concurrent processes can lead to a combinatorial explosion of the state space. By extension, parity games that encode model checking problems for concurrent processes suffer from the same phenomenon. *Partial-order reduction* (POR) techniques help combat the blowup. Several variants of POR exist, such as *ample sets* [26], *persistent sets* [8] and *stubborn sets* [30,31]. The current work is based on Valmari’s stubborn set theory as it can easily deal with nondeterminism [32].

3.1 Weak Stubborn Sets

Partial-order reduction relies on edge labels, here referred to as *events* and typically denoted with the letter j , to categorise the set of edges in a graph and determine independence of edges. In a typical application of POR, such events and edge labellings are deduced from a high-level syntactic description of the graph structure (see also Section 4). A *reduction function* subsequently uses these events when producing an equivalent *reduced* graph structure from the same high-level description. For now, we tacitly assume the existence of a set of events and edge labellings for parity games and refer to the resulting structures as *labelled parity games*.

Definition 2. A labelled parity game is a triple $L = (G, \mathcal{S}, \ell)$, where $G = (V, E, \Omega, \mathcal{P})$ is a parity game, \mathcal{S} is a non-empty set of events and $\ell : \mathcal{S} \rightarrow 2^E$ is an edge labelling.

For the remainder of this section, we fix an arbitrary labelled parity game $L = (G, \mathcal{S}, \ell)$. We write $s \xrightarrow{j} t$ whenever $s \rightarrow t$ and $(s, t) \in \ell(j)$. The same notation extends to longer executions $s \xrightarrow{j_1 \dots j_n} t$. We say an event j is enabled in a node s , notation $s \overset{j}{\rightarrow}$, if and only if there is a transition $s \xrightarrow{j} t$ for some t . The set of all enabled events in a node s is denoted with $enabled_G(s)$. An event j is *invisible* if and only if $s \xrightarrow{j} t$ implies $\mathcal{P}(s) = \mathcal{P}(t)$ and $\Omega(s) = \Omega(t)$. Otherwise, j is *visible*.

A *reduction function* indicates which edges are to be explored in each node, based on the events associated to the edges. Given some initial node \hat{s} , such a function induces a unique *reduced labelled parity game* as follows.

Definition 3. Given a node $\hat{s} \in V$ and a reduction function $r : V \rightarrow 2^{\mathcal{S}}$. The reduced labelled parity game induced by r and starting from \hat{s} is defined as $L_r = (G_r, \mathcal{S}, \ell_r)$, where $\ell_r(j) = \ell(j) \cap E_r$ and $G_r = (V_r, E_r, \Omega, \mathcal{P})$ is such that:

- $E_r = \{(s, t) \in E \mid \exists j \in r(s) : (s, t) \in \ell(j)\}$ is the transition relation under r ;
- $V_r = \{s \mid \hat{s} E_r^* s\}$ is the set of nodes reachable with E_r , where E_r^* is the reflexive transitive closure of E_r .

Note that a reduced labelled parity game is only well-defined when $r(s) \cap enabled_G(s) \neq \emptyset$ for every node $s \in V_r$; if this property does not hold, E_r is not total. Even if totality of E_r is guaranteed, the same node s may be won by different players in L and L_r if no restrictions are imposed on r . The following conditions on r , as we will show, are sufficient to ensure both. Below, we say an event j is a *key event* in s iff for all executions $s \xrightarrow{j_1 \dots j_n} s'$ such that $j_1 \notin r(s), \dots, j_n \notin r(s)$, we have $s' \xrightarrow{j}$. Key events are typically denoted j_{key} .

Definition 4. We say that a reduction function $r : V \rightarrow 2^{\mathcal{S}}$ is a weak stubborn set iff for all nodes $s \in V$, the following conditions hold¹:

¹ As noted before, the condition **D1** that we propose is stronger than the version in literature [30,33] since that one suffers from the *inconsistent labelling problem* [21] which also manifests itself in the parity game setting, see our technical report [24].

- D1** For all $j \in r(s)$ and $j_1 \notin r(s), \dots, j_n \notin r(s)$, if $s \xrightarrow{j_1} s_1 \xrightarrow{j_2} \dots \xrightarrow{j_n} s_n \xrightarrow{j} s'_n$, then there are nodes s', s_1, \dots, s'_{n-1} such that $s \xrightarrow{j} s' \xrightarrow{j_1} s'_1 \xrightarrow{j_2} \dots \xrightarrow{j_n} s'_n$. Furthermore, if j is invisible, then $s_i \xrightarrow{j} s'_i$ for every $1 \leq i < n$.
- D2w** $r(s)$ contains a key event in s .
- V** If $r(s)$ contains an enabled visible event, then it contains all visible events.
- I** If an invisible event is enabled, then $r(s)$ contains an invisible key event.
- L** For every visible event j , every cycle in the reduced game contains a node s' such that $j \in r(s')$.

Below, we also use (weak) stubborn set to refer to the set of events $r(s)$ in some node s . First, note that every key event, which we typically denote by j_{key} , in a node s is enabled in s , by taking $n = 0$ in **D2w**; this guarantees totality of E_r . Condition **D1** ensures that whenever an enabled event is selected for the stubborn set, it does not disable executions not in $r(s)$. A stubborn set can never be empty, due to **D2w**. In a traditional setting where POR is applied on a transition system, the combination of **D1** and **D2w** is sufficient to preserve deadlocks. Condition **V** enforces that either all visible events are selected for the stubborn set, or none are. Condition **L** prevents the so called *action-ignoring problem*, where a certain event is never selected for the stubborn set and ignored indefinitely. Combined, **I** and **L** preserve plays with invisible events only.

We use the example below to further illustrate the purpose of—and need for—conditions **V**, **I** and **L**. In particular, the example illustrates that the winning player in the original game and the reduced game might be different if one of these conditions is not satisfied.

Example 2. See the three parity games of Figure 1. From left to right, these games show a reduced game under a reduction function satisfying **D1** and **D2w** but not **V**, **I** or **L**, respectively. In each case, we start exploration from the node called \hat{s} , using the reduction function to follow the solid edges; consequently, the winning strategy σ_\diamond for player \diamond in the original game is lost. □

Note that the games in Figure 1 are from a subclass of parity games called *weak solitaire*, illustrating the need for the identified conditions even in restricted

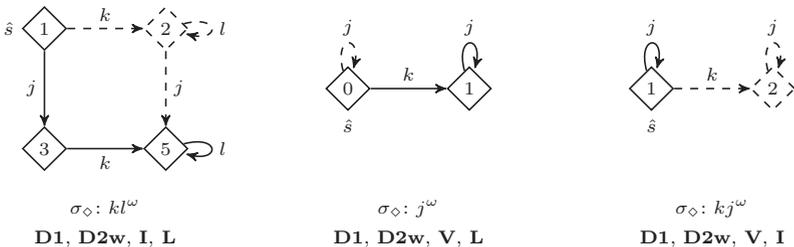


Fig. 1. Three games that show the winner is not necessarily preserved if we drop one of the conditions **V**, **I** or **L**, respectively. The dashed nodes and edges are present in the original game, but not in the reduced game. The edges taken from \hat{s} by the winning strategy for player \diamond in the original game are indicated below each game.

settings. A game is *weak* if the priorities along all its paths are non-decreasing, i.e., if $s \rightarrow t$ then $\Omega(s) \leq \Omega(t)$. A game is *solitaire* if only one player can make non-trivial choices. Weak solitaire games can encode the model checking of safety properties; solitaire games can capture logics such as LTL and ACTL*.

Before we argue for the correctness of our POR approach in the next section, we finish with a small example that illustrates how our approach improves over existing methods for branching time logics.

Example 3. The conditions **C1-C3** of Gerth *et al.* [7] preserve LTL_{-X} and are similar in spirit to our conditions. However, to preserve the branching structure, needed for preservation of CTL_{-X} , the following *singleton proviso* is introduced: **C4** Either $enabled_G(s) \subseteq r(s)$ or $|r(s)| = 1$.

This extra condition can severely impact the amount of reduction achieved: consider the following two processes, where $n \geq 1$ is some large natural number.



The cross product of these processes contains $(n + 1)^2$ states. In the initial state, neither $\{a_1, a'_1\}$ nor $\{b_1, b'_1\}$ is a valid stubborn set, due to **C4**. However, the labelled parity game constructed using these processes and the μ -calculus formula $\nu X.([\neg]X \wedge \mu Y.(\langle \neg \rangle Y \vee \langle a_n \rangle true))$, has a very similar shape that *can* be reduced by prioritising transitions that correspond to b_i or b'_i for some $1 \leq i \leq n$. Note that this formula cannot be represented in LTL; condition **C4** is therefore essential for the correctness. While several optimisations for CTL_{-X} model checking under POR are proposed in [19], unlike our approach, those optimisations only work for certain classes of CTL_{-X} formulas and not in general. \square

3.2 Correctness

Condition **D2w** suffices, as we already argued, to preserve totality of the transition relation of the reduced labelled parity game. Hence, we are left to argue that the reduced game preserves and reflects the winner of the nodes of the original game; this is formally claimed in Theorem 1. We do so by constructing a strategy in the reduced game that mimics the winning strategy in the original game. The plays that are consistent with these two strategies are then shown to be *stutter equivalent*, which suffices to preserve the winner.

Fix a labelled parity game $L = (G, \mathcal{S}, \ell)$, a node \hat{s} , a weak stubborn set r and the reduced labelled parity game $L_r = (G_r, \mathcal{S}, \ell_r)$ induced by r and \hat{s} . We assume r and \hat{s} are such that G_r has a finite state space. Below, ω is the set containing all natural numbers and the smallest infinite ordinal number.

Definition 5. Let $\pi = s_0s_1s_2\dots$ and $\pi' = t_0t_1t_2\dots$ be two paths in G . We say π and π' are *stutter equivalent*, notation $\pi \triangleq \pi'$, if and only if one of the following conditions holds:

- π and π' are both finite and there exists a non-decreasing partial function $f : \omega \rightarrow \omega$, with $f(0) = 0$ and $f(|\pi|-1) = |\pi'|-1$, such that for all $0 \leq i < |\pi|$ and $i' \in [f(i), f(i + 1))$, it holds that $\mathcal{P}(s_i) = \mathcal{P}(t_{i'})$ and $\Omega(s_i) = \Omega(t_{i'})$.

– π and π' are both infinite and there exists an unbounded, non-decreasing total function $f : \omega \rightarrow \omega$, with $f(0) = 0$, such that for all i and $i' \in [f(i), f(i+1))$, it holds that $\mathcal{P}(s_i) = \mathcal{P}(t_{i'})$ and $\Omega(s_i) = \Omega(t_{i'})$.

Lemma 1. All infinite stutter equivalent paths have the same winner.

In the lemmata below, we write \rightarrow_r to stress which transition must occur in G_r .

Lemma 2. Suppose $s_0 \xrightarrow{j_1} \dots \xrightarrow{j_n} s_n \xrightarrow{j} s'_n$ for $j_1 \notin r(s_0), \dots, j_n \notin r(s_0)$ and $j \in r(s_0)$. Then for some s'_0, \dots, s'_n , both $s_0 \xrightarrow{j} s'_0 \xrightarrow{j_1} \dots \xrightarrow{j_n} s'_n$ and $s_0 \dots s_n s'_n \triangleq s_0 s'_0 \dots s'_n$.

Lemma 3. Suppose $s_0 \xrightarrow{j_1} s_1 \xrightarrow{j_2} \dots$ such that $j_i \notin r(s_0)$ for every j_i occurring on this execution. Then, the following holds:

- If the execution ends in s_n , there exists a key event j_{key} , and nodes s'_0, \dots, s'_n such that $s_n \xrightarrow{j_{\text{key}}} s'_n$ and $s_0 \xrightarrow{j_{\text{key}}}_r s'_0 \xrightarrow{j_1} \dots \xrightarrow{j_n} s'_n$, and $s_0 \dots s_n \triangleq s_0 s'_0 \dots s'_n$.
- If the execution is infinite, there exists another execution $s_0 \xrightarrow{j_{\text{key}}}_r s'_0 \xrightarrow{j_1} s'_1 \xrightarrow{j_2} \dots$ for some key event j_{key} and $s_0 s_1 \dots \triangleq s_0 s'_0 s'_1 \dots$.

We remark that Lemma 3 also holds for reduced labelled parity games that have an infinite state space, but where all the events are finitely branching. The proof of correctness, viz., Theorem 1, uses the alternative executions described by Lemma 2 and 3. For full details, we refer to [24]; we here limit ourselves to sketching the intuition behind the application of these lemmata.

Example 4. The structure of Figure 2, in which parallel edges have the same label, visualises part of a game in which the solid edges labelled $j_1 j_2 j_3$ are part of a winning play for player \square . This play is mimicked by path that follows the edges $j_{\text{key}} j_2 j_1 j'_{\text{key}} j_3$, drawn with dashes. The new play reorders the events j_1, j_2 and j_3 according to the construction of Lemma 2 and introduces the key events j_{key} and j'_{key} according to the construction of Lemma 3. \square

The following theorem shows that partial-order reduction preserves the winning player in all nodes of the reduced game. Its proof is inspired by [30] and [2, Lemma 8.21], and uses the aforementioned lemmata.

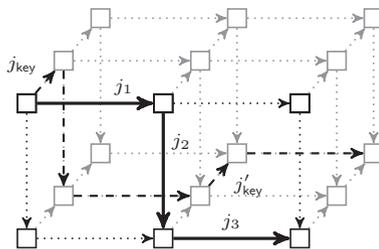


Fig. 2. Example of how j_1, j_2, j_3 is mimicked by introducing j_{key} and j'_{key} and moving j_2 to the front (dashed trace). Transitions that are drawn in parallel have the same label.

Theorem 1. *If G_r has a finite state space then it holds that for every node s in G_r , the winner of s in G_r is equal to the winner of s in G .*

3.3 Optimising D2w

The theory we have introduced identifies and exploits rectangular structures in the parity game. This is especially apparent in condition **D1**. However, parity games obtained from model checking problems also often contain triangular structures, due to the (sometimes implicit) nesting of conjunctions and disjunctions, as the following example demonstrates.

Example 5. Consider the process $(a \parallel b) \cdot c$, in which actions a and b are executed in (interleaved) parallel, and action c is executed upon termination of both a and b . The μ -calculus property $\mu X.([a]X \wedge [b]X \wedge \langle - \rangle true)$, also expressible in LTL, expresses that the action c must unavoidably be done within a finite number of steps; clearly this property holds true of the process. Below, the LTS is depicted on the left and a possible parity game encoding of our liveness property on this state space is depicted on the right. The edges in the labelled parity game that originate from the subformula $\langle - \rangle true$ are labelled with d .



Whereas the state space of the process can be reduced by prioritising a or b , the labelled parity game cannot be reduced due to the presence of a d -labelled edge in every node. For example, if s is the top-left node in the labelled parity game, then $r(s) = \{a, d\}$ violates condition **D1**, since the execution $s \xrightarrow{bd}$ exists, but $s \xrightarrow{db}$ does not. \square

In order to deal with games that contain triangular structures, we propose a condition that is weaker than **D2w**.

D2t There is an event $j \in r(s)$ such that for all $j_1 \notin r(s), \dots, j_n \notin r(s)$, if $s \xrightarrow{j_1} s_1 \xrightarrow{j_2} \dots \xrightarrow{j_n} s_n$, then either $s_n \xrightarrow{j}$ or there are nodes s'_i, s'_1, \dots, s'_n such that $s \xrightarrow{j} s' \xrightarrow{j_1} s'_1 \xrightarrow{j_2} \dots \xrightarrow{j_n} s'_n$ and for all i , $s_i = s'_i$ or $s_i \xrightarrow{j} s'_i$.

Theorem 1 holds even for reduction functions satisfying the weak stubborn set conditions in which condition **D2t** is used instead of condition **D2w**. The proof thereof resorts to a modified construction of a mimicking winning strategy that is based on Lemma 4, described below, instead of Lemma 3.

Lemma 4. *Let r be a reduction function satisfying conditions **D1**, **D2t**, **V**, **I** and **L**. Suppose $s_0 \xrightarrow{j_1} s_1 \xrightarrow{j_2} \dots$ such that $j_i \notin r(s_0)$ for every j_i occurring on this execution. Then, the following holds:*

- *If the execution ends in s_n , there exist a key event j_{key} and nodes s'_0, \dots, s'_n such that:*
 - $s_n \xrightarrow{j_{key}} s'_n$ or $s_n = s'_n$; and

- $s_0 \xrightarrow{j_{\text{key}}}_r s'_0 \xrightarrow{j_1} \dots \xrightarrow{j_n} s'_n$ and $s_0 \dots s_n \triangleq s_0 s'_0 \dots s'_n$.
- If the execution is infinite, there exists another execution $s_0 \xrightarrow{j_{\text{key}}}_r s'_0 \xrightarrow{j_1} s'_1 \xrightarrow{j_2} \dots$ and $s_0 s_1 \dots \triangleq s_0 s'_0 s'_1 \dots$.

We remark that the concepts of triangular and rectangular structures bear similarities to the concept of weak confluence from [9].

4 Parameterised Boolean Equation Systems

Parity games are used, among others, to solve *parameterised Boolean equation systems* (PBESs) [10], which, in turn, are used to answer, *e.g.*, first-order modal μ -calculus model checking problems [5]. In the remainder of this paper, we show how to apply POR in the context of solving a PBES (and, hence, the encoded decision problem). We first introduce PBESs and show how they induce labelled parity games.

Parameterised Boolean equation systems are sequences of fixed point equations over predicate formulae, *i.e.*, first-order logic formulae with second order variables. A PBES is given in the context of an abstract data type, which is used to reason about data. Non-empty data sorts of the abstract data type are typically denoted with the letters D and E . The corresponding semantic domains are \mathbb{D} and \mathbb{E} . We assume that sorts B and N represent the Booleans and the natural numbers respectively, and have \mathbb{B} and \mathbb{N} as semantic counterpart. The set of data variables is \mathcal{V} , and its elements are usually denoted with d and e . To interpret expressions with variables, we use a *data environment* δ , which maps every variable in \mathcal{V} to an element of the corresponding sort. The semantics of an expression f in the context of such an environment is denoted $\llbracket f \rrbracket \delta$. For instance, $\llbracket x < 2 + y \rrbracket \delta$ holds true iff $\delta(x) < 2 + \delta(y)$. To update an environment, we use the notation $\delta[v/d]$, which is defined as $\delta[v/d](d) = v$ and $\delta[v/d](d') = \delta(d')$ for all variables $d \neq d'$.

For lack of space, we only consider PBESs in *standard recursive form* [22,23], a normal form in which each right-hand side of an equation is a *guarded* formula instead of an arbitrary (monotone) predicate formula. We remark that a PBES can be rewritten to SRF in linear time, while the number of equations grows linearly in the worst case [23, Proposition 2].

Let \mathcal{X} be a countable set of predicate variables. In the exposition that follows we assume for the sake of simplicity (but without loss of generality) that all predicate variables $X \in \mathcal{X}$ are of type D . We permit ourselves the use of non-uniformly typed predicate variables in our example.

Definition 6. A guarded formula ϕ is a disjunctive or conjunctive formula of the form:

$$\bigvee_{j \in J} \exists e_j : E_j. f_j \wedge X_j(g_j) \text{ or } \bigwedge_{j \in J} \forall e_j : E_j. f_j \Rightarrow X_j(g_j)$$

where J is an index set, each f_j is a Boolean expression, referred to as guard, every e_j is a (bound) variable of sort E_j , each g_j is an expression of type D and

each X_j is a predicate variable of type D . A guarded formula ϕ is said to be total if for each data environment δ , there is a $j \in J$ and $v \in \mathbb{E}_j$ such that $\llbracket f_j \rrbracket \delta[v/e_j]$ holds true.

The denotational semantics of a guarded formula is given in the context of a data environment δ for interpreting data expressions and a predicate environment $\eta : \mathcal{X} \rightarrow 2^{\mathbb{D}}$, yielding an interpretation of $X_j(g_j)$ as the truth value $\llbracket g_j \rrbracket \delta \in \eta(X_j)$. Given a predicate environment and a data environment, a guarded formula induces a monotone operator on the complete lattice $(2^{\mathbb{D}}, \subseteq)$. By Tarski's theorem, least (μ) and greatest (ν) fixed points of such operators are guaranteed to exist.

Definition 7. A parameterised Boolean equation in SRF is an equation that has the shape $(\mu X(d:D) = \phi(d))$ or $(\nu X(d:D) = \phi(d))$, where $\phi(d)$ is a total guarded formula in which d is the only free data variable. A parameterised Boolean equation system in SRF is a sequence of parameterised Boolean equations in SRF, in which no two equations have the same left-hand side variable.

Henceforward, let $\mathcal{E} = (\sigma_1 X_1(d:D) = \varphi_1(d)) \dots (\sigma_n X_n(d:D) = \varphi_n(d))$ be a fixed, arbitrary PBES in SRF, where $\sigma_i \in \{\mu, \nu\}$. The set of bound predicate variables of \mathcal{E} , denoted $\text{bnd}(\mathcal{E})$, is the set $\{X_1, \dots, X_n\}$. If the predicate variables occurring in the guarded formulae $\varphi_i(d)$ of \mathcal{E} are taken from $\text{bnd}(\mathcal{E})$, then \mathcal{E} is said to be closed; we only consider closed PBESs. Every bound predicate variable is assigned a rank, where $\text{rank}_{\mathcal{E}}(X_i)$ is the number of alternations in the sequence of fixpoint symbols $\nu\sigma_1\sigma_2 \dots \sigma_i$. Observe that $\text{rank}_{\mathcal{E}}(X_i)$ is even iff $\sigma_i = \nu$. We use the function $\text{op}_{\mathcal{E}} : \text{bnd}(\mathcal{E}) \rightarrow \{\vee, \wedge\}$ to indicate for each predicate variable in \mathcal{E} whether the associated equation is disjunctive or conjunctive. As a notational convenience, we write J_i to refer to the index set of the guarded formula $\varphi_i(d)$, and we assume that the index sets are disjoint for different equations.

The standard denotational fixed point semantics of a closed PBES associates a subset of \mathbb{D} to each bound predicate variable (*i.e.*, their meaning is independent of the predicate environment used to interpret guarded formulae). For details of the standard denotational fixed point semantics of a PBES we refer to [10]. We forego the denotational semantics and instead focus on the (provably equivalent, see *e.g.* [23,6]) game semantics of a PBES in SRF.

Definition 8. The solution to \mathcal{E} is a mapping $\llbracket \mathcal{E} \rrbracket : \text{bnd}(\mathcal{E}) \rightarrow 2^{\mathbb{D}}$, defined as $\llbracket \mathcal{E} \rrbracket(X_i) = \{v \in \mathbb{D} \mid (X_i, v) \text{ is won by } \diamond \text{ in } G_{\mathcal{E}}\}$, where $X_i \in \text{bnd}(\mathcal{E})$ and $G_{\mathcal{E}}$ is the parity game associated to \mathcal{E} . The game $G_{\mathcal{E}} = (V, E, \Omega, \mathcal{P})$ is defined as:

- $V = \text{bnd}(\mathcal{E}) \times \mathbb{D}$ is the set of nodes;
- E is the edge relation, satisfying $(X_i, v) \rightarrow (X_j, w)$ for given $X_i, j \in J_i, v$ and w if and only if for some δ , both $\llbracket f_j \rrbracket \delta[v/d]$ and $w = \llbracket g_j \rrbracket \delta[v/d]$ hold;
- $\Omega((X_i, v)) = \text{rank}_{\mathcal{E}}(X_i)$; and
- $\mathcal{P}((X_i, v)) = \diamond$ iff $\text{op}_{\mathcal{E}}(X_i) = \vee$.

Note that the parity game $G_{\mathcal{E}}$ may have an infinite state space when \mathbb{D} is infinite. In practice, we are often interested in the part of the parity game that is reachable from some initial node (X, v) ; this is often (but not always) finite. This is illustrated by the following example.

Example 6. Consider the following PBES in SRF:

$$\begin{aligned}
 (\nu X(b:B) &= (b \wedge X(\text{false})) \vee \exists n:N. n \leq 2 \wedge Y(b, \text{if}(b, n, 0))) \\
 (\mu Y(b:B, n:N) &= \text{true} \Rightarrow Y(\text{false}, 0))
 \end{aligned}$$

The six nodes in the parity game which are reachable from (X, true) are depicted in Figure 3. The horizontally drawn edges all stem from the clause $\exists n:N. n \leq 2 \wedge Y(b, \text{if}(b, n, 0))$. Vertical edges stem from the clause $b \wedge X(\text{false})$ (on the left) or the clause $\text{true} \Rightarrow Y(\text{false}, 0)$ (on the right). The selfloop also stems from the clause $\text{true} \Rightarrow Y(\text{false}, 0)$. Player \square wins all nodes in this game, and thus $\text{true} \notin \llbracket \mathcal{E} \rrbracket(X)$. \square

As suggested by the above example, each edge is associated to (at least) one clause in \mathcal{E} . Consequently, we can use the index sets J_i to event-label the edges emanating from nodes associated with the equation for X_i . We denote the set of all events in \mathcal{E} by $\text{evt}(\mathcal{E})$, defined as $\text{evt}(\mathcal{E}) = \bigcup_{X_i \in \text{bnd}(\mathcal{E})} J_i$. Event $j \in J_i$ is *invisible* if $\text{rank}_{\mathcal{E}}(X_i) = \text{rank}_{\mathcal{E}}(X_j)$ and $\text{op}_{\mathcal{E}}(X_i) = \text{op}_{\mathcal{E}}(X_j)$, and *visible* otherwise.

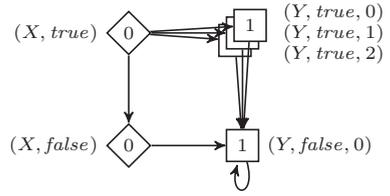


Fig. 3. Reachable part of the parity game underlying the PBES of Example 6, when starting from node (X, true) .

Definition 9. Let $G_{\mathcal{E}}$ be the parity game associated to \mathcal{E} . The labelled parity game associated to \mathcal{E} is the structure $(G_{\mathcal{E}}, \text{evt}(\mathcal{E}), \ell)$, where $G_{\mathcal{E}}$ is as defined in Def. 8, and, for $j \in J_i$, $\ell(j)$ is defined as the set $\{ \langle (X_i, v), (X_j, w) \rangle \in E \mid \llbracket f_j \rrbracket \delta[v/d]$ holds true and $w = \llbracket g_j \rrbracket \delta[v/d]$ for some δ }.

5 PBES Solving Using POR

A consequence of the partial-order reduction theorem is that a reduced parity game suffices for computing the truth value to $X(e)$ for a given PBES \mathcal{E} with $X \in \text{bnd}(\mathcal{E})$. However, **D1**, **D2w/D2t** and **L** are conditions on the (reduced) state space as a whole and, hence, hard to check locally. We therefore approximate these conditions in such a way that we can construct a stubborn set *on-the-fly*.

From hereon, let \mathcal{E} be a PBES in SRF and (G, \mathcal{S}, ℓ) , with $G = (V, E, \Omega, \mathcal{P})$, its labelled parity game. The most common local condition for **L** is the *stack proviso* \mathbf{L}^S [26]. This proviso assumes that the state space is explored with *depth-first search* (DFS), and it uses the *Stack* that stores unexplored nodes to determine whether a cycle is being closed. If so, the node will be *fully expanded*, i.e., $r(s) = \mathcal{S}$.

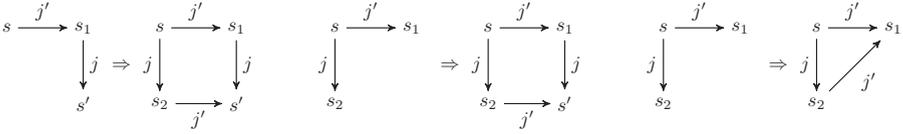
\mathbf{L}^S For all nodes $s \in V_r$, either $\text{succ}_{G_r}(s) \cap \text{Stack} = \emptyset$ or $r(s) = \mathcal{S}$.

Locally approximating conditions **D1** and **D2w** requires a static analysis of the PBES. For this, we draw upon ideas from [17] and extend these to properly deal with non-determinism. To reason about which events are independent, we rely on the idea of *accordance*.

Definition 10. Let $j, j' \in \mathcal{S}$. We define the accordance relations *DNL*, *DNS*, *DNT* and *DNA* on \mathcal{S} as follows:

- j left-accords with j' if for all nodes $s, s' \in V$, if $s \xrightarrow{j'} s'$, then also $s \xrightarrow{j} s'$. If j does not left-accord with j' , we write $(j, j') \in \text{DNL}$.
- j square-accords with j' if for all nodes $s, s_1, s_2 \in V$, if $s \xrightarrow{j} s_1$ and $s \xrightarrow{j'} s_2$, then for some $s' \in V$, $s_1 \xrightarrow{j'} s'$ and $s_2 \xrightarrow{j} s'$. If j does not square-accord with j' we write $(j, j') \in \text{DNS}$.
- j triangle-accords with j' if for all nodes $s, s_1, s_2 \in V$, if $s \xrightarrow{j} s_1$ and $s \xrightarrow{j'} s_2$, then $s_2 \xrightarrow{j} s_1$. If j does not triangle-accord with j' we write $(j, j') \in \text{DNT}$.
- j accords with j' if j square-accords or triangle-accords with j' . If j does not accord with j' we write $(j, j') \in \text{DNA}$.

Note that *DNL* and *DNT* are not necessarily symmetric. An illustration of the left-according, square-according and triangle-according conditions is given below.



Accordance relations safely approximate the independence of events. The dependence of events, required for satisfying **D2w** can be approximated using Godefroid’s *necessary enabling sets* [8].

Definition 11. Let j be an event that is disabled in some node s . A necessary-enabling set (*NES*) for j in s is any set $NES_s(j) \subseteq \mathcal{S}$ such that for every execution $s \xrightarrow{j_1 \dots j_n j}$ there is at least one j_i such that $j_i \in NES_s(j)$.

For every node and event there might be more than one NES. In particular, every superset of a NES is also a NES. A larger-than-needed NES may, however, have a negative impact on the reduction that can be achieved. In a PBES with multiple parameters per predicate variable, computing a NES can be done by determining which parameters influence the validity of guards f_j and which parameters are changed in the update functions g_j . A more accurate NES may be computed using techniques to extract a control flow from a PBES [15].

The following lemmata show how the accordance relations and necessary-enabling set can be used to implement conditions **D1**, **D2w** and **D2t**, respectively. A combination of Lemma 5 and 6 in a deterministic setting appeared as Lemma 1 in [17]. Note that as a notational convention we write $R(j)$ to denote the projection $\{j' \mid (j, j') \in R\}$ of a binary relation.

Lemma 5. A reduction function r satisfies **D1** in node $s \in V$ if for all $j \in r(s)$:
 – if j is disabled in s , then $NES_s(j) \subseteq r(s)$ for some NES_s ; and
 – if j is enabled in s , then $\text{DNL}(j) \subseteq r(s)$.

Lemma 6. A reduction function r satisfies **D2w** in a node $s \in V$ if there is an enabled event $j \in r(s)$ such that $\text{DNS}(j) \subseteq r(s)$.

Lemma 7. A reduction function r satisfies **D2t** in a node s if there is an enabled event $j \in r(s)$ such that $\text{DNA}(j) \subseteq r(s)$.

More reduction can be achieved if a PBES is partly or completely ‘deterministic’, in which case some of the conditions can be relaxed. We say that an event j is *deterministic*, denoted by $\text{det}(j)$, if for all nodes $t, t', t'' \in V$, if $t \xrightarrow{j} t'$ and $t \xrightarrow{j} t''$, then also $t' = t''$. This means event-determinism can be characterised as follows:

$$\text{det}(j) \text{ iff } \llbracket f_j \rrbracket \delta \text{ and } \llbracket f_j \rrbracket \delta' \text{ implies } \llbracket g_j \rrbracket \delta = \llbracket g_j \rrbracket \delta' \text{ for all } \delta, \delta' \text{ with } \delta(d) = \delta'(d).$$

The following lemma specialises Lemma 5 and shows how knowledge of deterministic events can be applied to potentially improve the reduction.

Lemma 8. *A reduction function r satisfies **D1** in a node s if for all $j \in r(s)$:*

- *if j is disabled in s , then $\text{NES}_s(j) \subseteq r(s)$ for some NES_s ; and*
- *if $\text{det}(j)$ and j is enabled in s , then $\text{DNS}(j) \subseteq r(s)$ or $\text{DNL}(j) \subseteq r(s)$.*
- *if $\neg \text{det}(j)$ and j is enabled in s , then $\text{DNL}(j) \subseteq r(s)$.*

Since relations DNS and DNL are incomparable we cannot decide *a priori* which should be used for deterministic events. However, Lemma 8 permits choosing one of the accordance sets on-the-fly. This choice can be made based on a heuristic function, similar to the function for NESs proposed in [17].

6 Experiments

We implemented the ideas from the previous section in a prototype tool, called `pbessor`, as part of the mCRL2 toolset [5]; it is written in C++. Our tool converts a given input PBES to a PBES in SRF, runs a static analysis to compute the accordance relations (see Section 5), and uses a depth-first exploration to compute the parity game underlying the PBES in SRF. The static analysis relies on an external SMT solver (we use Z3 in our experiments). To limit the amount of static analysis required and to improve the reduction, the implementation contains a rudimentary way of identifying whether the same event occurs in multiple PBES equations. Experiments are conducted on a machine with an Intel Xeon 6136 CPU @ 3 GHz, running mCRL2 with Git commit hash `dd36f98875`.

To measure the effectiveness of our implementation, we analysed the following mCRL2 models²: Anderson’s mutual exclusion protocol [1], the dining philosophers problem, the gas station problem [11], Hesselink’s handshake register [12], Le Lann’s leader election protocol [18], Milner’s Scheduler [20] and the Krebs cycle of ATP production in biological cells (model inspired by [25]). Most of these models are scalable. Each model is subjected to one or more requirements phrased as mCRL2’s first-order modal μ -calculus formulae. Where possible, Table 1 provides a CTL* formula that captures the essence of the requirement.

We analyse the effectiveness of our partial-order reduction technique by measuring the reduction of the size of the state space, and the time that is required to generate the state space. Since the static analysis that is conducted can require a non-negligible amount of time, we pay close attention to the various forms of static analysis that can be conducted. In particular, we compare the total time and effectiveness (in terms of reduction) of running the following static analysis:

² The models are archived online at <https://doi.org/10.5281/zenodo.3602969>.

Table 1. Runtime (analysis + exploration; in seconds) and number of states when exploring either the full state space or the reduced state space, for four different static analysis approaches. Figures printed in boldface indicate which of the additional static analyses is able to achieve the largest reduction over ‘basic’ (if any).

model	property	full		basic		+DNL		+NES		+D2t	
		nodes	time	nodes	time	nodes	time	nodes	time	nodes	time
gas station.c3	$\exists \diamond \text{accept}$	1 197	0.14	1 077	0.98	1 077	2.48	1 077	1.87	735	1.62
gas station.c3	$\exists \square \exists \diamond \text{pumping}$	1 261	0.15	967	0.98	967	2.61	967	1.99	967	1.72
gas station.c3	no deadlock	1 197	0.18	735	0.95	735	2.52	735	2.04	735	1.52
scheduler8	no deadlock	3 073	0.29	34	0.19	34	0.70	34	0.51	34	0.35
scheduler10	no deadlock	15 361	1.65	42	0.25	42	0.90	42	0.65	42	0.42
anderson.5	$\forall \diamond \text{cs}$	23 597	4.59	2 957	2.85	2 957	6.47	2 957	3.89	2 957	4.61
hesslink	cache consistency	91 009	5.28	82 602	8.19	83 602	12.12	81 988	9.00	71 911	8.51
dining10	no deadlock	154 451	17.90	4 743	0.76	4 743	1.61	4 743	1.42	4 743	1.02
krebs.3	$\forall \diamond \text{energy}$	238 877	24.38	232 273	24.59	232 273	25.62	209 345	21.73	232 273	24.42
gas station.c6	$\exists \diamond \text{accept}$	186 381	38.00	150 741	40.55	150 741	45.50	150 741	43.16	75 411	21.40
gas station.c6	$\exists \square \exists \diamond \text{pumping}$	192 700	38.63	114 130	27.35	114 130	31.42	114 130	30.49	114 130	29.74
gas station.c6	no deadlock	186 381	42.50	75 411	21.03	75 411	24.88	75 411	24.01	75 411	23.02
scheduler14	no deadlock	344 065	53.14	58	0.37	58	1.31	58	0.97	58	0.61
hesslink	$\forall \square (wr \Rightarrow \exists \diamond \text{fin})$	1 047 233	61.02	1 013 441	82.44	1 013 441	86.49	1 013 441	84.59	791 273	61.56
hesslink	$\forall \square (wr \Rightarrow \forall \diamond \text{fin})$	1 047 232	70.14	791 320	64.05	791 374	66.53	749 936	62.98	791 268	67.59
krebs.4	$\forall \diamond \text{energy}$	1 047 406	124.30	971 128	117.38	971 128	117.41	843 349	101.51	971 128	117.41
lann.5	consistent data	818 104	142.38	818 104	170.18	818 104	175.87	818 104	177.78	761 239	155.22
anderson.5	no deadlock	689 901	142.63	257 944	73.62	257 672	79.91	257 711	78.67	257 918	76.47
lann.5	no data loss	1 286 452	199.74	453 130	73.28	453 130	77.31	453 130	74.40	453 130	75.52
dining10	$\forall \square \forall \diamond \text{eat}$	1 698 951	225.10	101 185	12.37	101 056	13.55	101 238	13.01	101 022	12.69
anderson.7	$\forall \diamond \text{cs}$	3 964 599	1 331.91	124 707	63.83	124 707	73.87	124 707	68.67	124 707	69.68

- computing left-accordance (*DNL*) vs. over-approximating it with all events.
- computing a NES vs. over-approximating it with the set of all events *S*.
- using **D2w** vs. the use of **D2t** (*i.e.*, use Lemma 6 vs. Lemma 7);

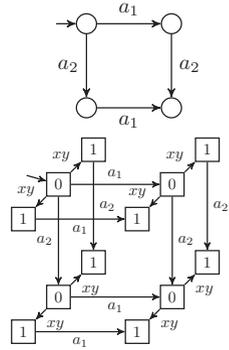
As a baseline for comparisons, we take a basic static analysis (over-approximated *DNL*, over-approximated NES, **D2w**), see column ‘basic’ in Table 1. In order to guarantee termination of the static analysis phase, we set a timeout of 200ms per formula that is sent to the solver. Table 1 reports on the statistics we obtained for exploring the full state space and the four possible POR configurations described above; the table is sorted with respect to the time needed for a full exploration. The time we list consists of the time needed to conduct the analysis plus the time needed for the exploration.

For most small instances, the time required for static analysis dominates any speed-up gained by the state space reduction. When the state spaces are larger, achieving a speed-up becomes more likely, while the highest overhead suffered by ‘basic’ is 55% (Hesslink, cache consistency). Significant reduction can be achieved even for non-trivial properties, such as ‘lann.5’ with ‘no data loss’. Scheduler is an extreme case: its processes have very few dependencies, leading to an exponential reduction, both in terms of the state space size and in terms of time. In several cases, the use of a NES or **D2t** brings extra reduction (highlighted in bold). Moreover, the extra time required to conduct the additional analysis seems limited. The use of *DNL*, on the other hand, never pays off in our experiments; it even results in a slightly larger state space in two cases.

We note that there are also models, not listed in Table 1, where our static analysis does not yield any useful results and no reduction is achieved. Even if in such cases a reduction would be possible in theory, the current static analysis engines are unable to deal with the more complex data types often used in such models; *e.g.*, recursively defined lists or infinite sets, represented symbolically with higher-order constructions. This calls for further investigations into static analysis theories that can effectively deal with complex data.

Finally, we point out that in the case of, *e.g.*, the dining philosophers problem, the relative reduction under the ‘no deadlock’ property is much better than under the ‘ $\forall \square \forall \diamond eat$ ’ property. This demonstrates the impact properties can have on the reductions achievable, and it also points at a phenomenon we have not stressed in the current work, *viz.*, the impact of identifying events on the reductions achievable. We explain the phenomenon in the following example.

Example 7. Consider the LTS and the parity game on the right. The parity game encodes the property $\nu X.([\neg]X \wedge \forall i. \mu Y.([\bar{a}_i]Y \wedge \langle - \rangle true))$, which is equivalent to $\forall \square \diamond a_i$, on this LTS. The event xy represents the transition from fixpoint X into Y , which does not involve an action from the LTS. Note that the complete state space is encoded in the fixpoint X . Due to the absence of some transitions in the part of the state space encoded in fixpoint Y , neither a_1 nor a_2 is according with xy . Hence, the only stubborn set in the initial node is $\{a_1, a_2, xy\}$, which yields no reduction. \square



Improving the event identification procedure can yield more reduction. For instance, if, for each i (bound in the universal quantifier), a different event xy_i is created, then both a_1, xy_2 and a_2, xy_1 will be according. If we disregard the visibility of xy_1 and xy_2 , four nodes can be eliminated.

7 Conclusion

We have presented an approach for applying partial-order reduction on parity games. This has two main advantages over POR applied on LTSs or Kripke structures: our approach supports the full modal μ -calculus, not just a fragment thereof, and the potential for reduction is greater, because we do not require a singleton proviso. Furthermore, we have shown how the ideas can be implemented with PBESs as a high-level representation. In future work, we aim to gain more insight into the effect of identifying events across PBES equations in several ways. We also want to investigate the possibility of solving a reduced parity game while it is being constructed. In certain cases, one may be able to decide the winner of the original game from this partial solution.

References

1. Anderson, T.E.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel & Distributed Systems* **1**(1), 6–16 (1990). <https://doi.org/10.1109/71.80120>
2. Baier, C., Katoen, J.P.: *Principles of model checking*. MIT Press (2008)
3. Bønneland, F.M., Jensen, P.G., Larsen, K.G., Muñoz, M.: Partial Order Reduction for Reachability Games. In: *CONCUR 2019*. vol. 140, pp. 23:1–23:15 (2019). <https://doi.org/10.4230/LIPIcs.CONCUR.2019.23>
4. Bønneland, F.M., Jensen, P.G., Larsen, K.G., Müniz, M., Srba, J.: Stubborn Set Reduction for Two-Player Reachability Games. *arXiv:1912.09875* (2019)
5. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, J.W., Wijs, A.W., Willemse, T.A.C.: The mCRL2 Toolset for Analysing Concurrent Systems: Improvements in Expressivity and Usability. In: *TACAS 2019*. LNCS, vol. 11428, pp. 21–39 (2019). https://doi.org/10.1007/978-3-030-17465-1_2
6. Cranen, S., Luttkik, B., Willemse, T.A.C.: Proof graphs for parameterised Boolean equation systems. In: *CONCUR 2013*. LNCS, vol. 8052, pp. 470–484 (2013). https://doi.org/10.1007/978-3-642-40184-8_33
7. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A Partial Order Approach to Branching Time Logic Model Checking. *Information and Computation* **150**(2), 132–152 (1999). <https://doi.org/10.1006/inco.1998.2778>
8. Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems*, LNCS, vol. 1032. Springer (1996). <https://doi.org/10.1007/3-540-60761-7>
9. Groote, J.F., Sellink, M.P.A.: Confluence for process verification. *Theoretical Computer Science* **170**(1-2), 47–81 (1996). [https://doi.org/10.1016/s0304-3975\(96\)00175-2](https://doi.org/10.1016/s0304-3975(96)00175-2)
10. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems. *Theoretical Computer Science* **343**(3), 332–369 (2005). <https://doi.org/10.1016/j.tcs.2005.06.016>
11. Heimbald, D., Luckham, D.: Debugging ada tasking programs. *IEEE Software* **2**(2), 47–57 (1985). <https://doi.org/10.1109/MS.1985.230351>
12. Hesselink, W.H.: Invariants for the construction of a handshake register. *Inf. Process. Lett.* **68**(4), 173–177 (1998). [https://doi.org/10.1016/S0020-0190\(98\)00158-6](https://doi.org/10.1016/S0020-0190(98)00158-6)
13. Ip, C.N., Dill, D.L.: Better verification through symmetry. *Formal Methods in System Design* **9**(1-2), 41–75 (1996). <https://doi.org/10.1007/BF00625968>
14. Kan, S., Huang, Z., Chen, Z., Li, W., Huang, Y.: Partial order reduction for checking LTL formulae with the next-time operator. *Journal of Logic and Computation* **27**(4), 1095–1131 (2017). <https://doi.org/10.1093/logcom/exw004>
15. Keiren, J.J.A., Wesselink, J.W., Willemse, T.A.C.: Liveness Analysis for Parameterised Boolean Equation Systems. In: *ATVA 2014*. LNCS, vol. 8837, pp. 219–234 (2014). https://doi.org/10.1007/978-3-319-11936-6_16
16. Kozen, D.: Results on the propositional μ -calculus. *Theoretical Computer Science* **27**(3), 333–354 (1982). [https://doi.org/10.1016/0304-3975\(82\)90125-6](https://doi.org/10.1016/0304-3975(82)90125-6)
17. Laarman, A., Pater, E., van de Pol, J., Hansen, H.: Guard-based partial-order reduction. *STTT* **18**(4), 427–448 (2016). <https://doi.org/10.1007/s10009-014-0363-9>
18. Lann, G.L.: *Distributed systems - towards a formal approach*. In: *IFIP*, 1977. pp. 155–160 (1977)
19. Liebke, T., Wolf, K.: Taking Some Burden Off an Explicit CTL Model Checker. In: *Petri Nets 2019*. LNCS, vol. 11522, pp. 321–341 (2019). https://doi.org/10.1007/978-3-030-21571-2_18

20. Milner, R.: A Calculus of Communicating Systems, LNCS, vol. 92. Springer (1980)
21. Neele, T., Valmari, A., Willemse, T.A.C.: The Inconsistent Labelling Problem of Stutter-Preserving Partial-Order Reduction. In: FoSSaCS 2020. LNCS, vol. 12077 (2020). https://doi.org/10.1007/978-3-030-45231-5_25
22. Neele, T., Willemse, T.A.C., Groote, J.F.: Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting. In: FACS 2018. LNCS, vol. 11222, pp. 216–236 (2018). https://doi.org/10.1007/978-3-030-02146-7_11
23. Neele, T., Willemse, T.A.C., Groote, J.F.: Finding compact proofs for infinite-data parameterised Boolean equation systems. *Science of Computer Programming* **188**, 102389 (2020). <https://doi.org/10.1016/j.scico.2019.102389>
24. Neele, T., Willemse, T.A.C., Wesselink, W.: Partial-Order Reduction for Parity Games with an Application on Parameterised Boolean Equation Systems (Technical Report). Tech. rep., Eindhoven University of Technology (2019)
25. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN 2007. LNCS, vol. 4595, pp. 263–267 (2007). https://doi.org/10.1007/978-3-540-73370-6_17
26. Peled, D.: All from One, One for All: on Model Checking Using Representatives. In: CAV 1993. LNCS, vol. 697, pp. 409–423 (1993). https://doi.org/10.1007/3-540-56922-7_34
27. Peled, D.: Combining partial order reductions with on-the-fly model-checking. *FMSD* **8**(1), 39–64 (1996). <https://doi.org/10.1007/BF00121262>
28. Ramakrishna, Y.S., Smolka, S.A.: Partial-Order Reduction in the Weak Modal Mu-Calculus. In: CONCUR 1997. LNCS, vol. 1243, pp. 5–24 (1997). https://doi.org/10.1007/3-540-63141-0_2
29. Siegel, S.F.: What’s Wrong with On-the-Fly Partial Order Reduction. In: CAV 2019. LNCS, vol. 11562, pp. 478–495 (2019). https://doi.org/10.1007/978-3-030-25543-5_27
30. Valmari, A.: A Stubborn Attack on State Explosion. *Formal Methods in System Design* **1**(4), 297–322 (1992). <https://doi.org/10.1007/BF00709154>
31. Valmari, A.: The state explosion problem. In: ACPN 1996. LNCS, vol. 1491, pp. 429–528 (1996). https://doi.org/10.1007/3-540-65306-6_21
32. Valmari, A.: Stubborn Set Methods for Process Algebras. In: POMIV 1996. DIMACS, vol. 29, pp. 213–231 (1997). <https://doi.org/10.1090/dimacs/029/12>
33. Valmari, A., Hansen, H.: Stubborn Set Intuition Explained. *ToPNoC* **10470**(12), 140–165 (2017). https://doi.org/10.1007/978-3-662-55862-1_7

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

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





Polynomial Identification of ω -Automata ^{*}

Dana Angluin¹ , Dana Fisman² , and Yaara Shoval²

¹ Yale University, New Haven, CT, USA

² Ben-Gurion University, Be'er Sheva, Israel

Abstract. We study identification in the limit using polynomial time and data for models of ω -automata. On the negative side we show that non-deterministic ω -automata (of types Büchi, coBüchi, Parity or Muller) can not be polynomially learned in the limit. On the positive side we show that the ω -language classes \mathbb{IB} , \mathbb{IC} , \mathbb{IP} , and \mathbb{IM} that are defined by deterministic Büchi, coBüchi, parity, and Muller acceptors that are isomorphic to their right-congruence automata (that is, the right congruences of languages in these classes are fully informative) are identifiable in the limit using polynomial time and data. We further show that for these classes a characteristic sample can be constructed in polynomial time.

Keywords: identification in the limit, characteristic sample, ω -regular.

1 Introduction

With the growing success of machine learning in efficiently solving a wide spectrum of problems, we are witnessing an increased use of machine learning techniques in formal methods for system design. One thread in recent literature uses general purpose machine learning techniques for obtaining more efficient verification/synthesis algorithms. Another thread, following the *automata theoretic approach to verification* [33,21] works on developing grammatical inference algorithms for verification and synthesis purposes. *Grammatical inference* (aka *automata learning*) refers to the problem of automatically inferring from examples a finite representation (e.g. an automaton, a grammar, or a formula) for an unknown language. The term *model learning* [31] was coined for the task of learning an automaton model for an unknown system. A large body of works has developed learning techniques for different automata types (e.g. I/O automata [1], register automata [20], symbolic automata [14], ω -automata [7], and program automata [25]) and has shown its usability in a diverse range of tasks.³

In grammatical inference, the learning algorithm does not learn a *language*, but rather a finite *representation* of it. Complexity of learning algorithms may

^{*} This research was supported by grant 2016239 from the United States – Israel Binational Science Foundation (BSF).

³ E.g., tasks such as black-box checking [28], specification mining [2], assume-guarantee reasoning [13], regular model checking [18], learning verification fixed-points [32], learning interfaces [27], analyzing botnet protocols [12] or smart card readers [10], finding security bugs [10], error localization [11], and code refactoring [26,29].

vary greatly by switching representations. For instance, if one wishes to learn regular languages, she may consider representations using deterministic finite automata (DFAs), non-deterministic finite automata (NFAs), regular expressions, linear grammars etc. Since the translation results between two such formalisms are not necessarily polynomial, a polynomial learnability result for one representation does not necessarily imply a polynomial learnability result for another representation. Let \mathbb{C} be a class of representations \mathcal{C} with a size measure $size(\mathcal{C})$ (e.g. for DFAs the size measure can be the number of states in the minimal automaton). We extend $size(\cdot)$ to the languages recognized by representations in \mathbb{C} by defining $size(L)$ to be the minimum of $size(\mathcal{C})$ over all \mathcal{C} representing L . In this paper we restrict attention to automata representations, namely, *acceptors*.

There are various learning paradigms considered in the grammatical inference literature, roughly classified into *passive* and *active*. We mention here the two central ones. In *passive learning* the model of *learning from finite data* refers to the following problem: given a finite sample $T \subseteq \Sigma^* \times \{0, 1\}$ of labeled words, a learning algorithm \mathbf{A} should return an acceptor \mathcal{C} that agrees with the sample T . That is, for every $(w, l) \in T$ the following holds: $w \in \llbracket \mathcal{C} \rrbracket$ iff $l = 1$ (where $\llbracket \mathcal{C} \rrbracket$ is the language accepted by \mathcal{C}). The class \mathbb{C} is *identifiable in the limit using polynomial time and data* if and only if there exists a polynomial time algorithm \mathbf{A} that takes as input a labeled sample T and outputs an acceptor $\mathcal{C} \in \mathbb{C}$ that is consistent with T , and \mathbf{A} also satisfies the following condition. If L is any language recognized by an automaton from class \mathbb{C} , then there exists a labeled sample T_L consistent with L of length bounded by a polynomial in $size(L)$, and for any labeled sample T consistent with L such that $T_L \subseteq T$, on input T the algorithm \mathbf{A} produces an acceptor \mathcal{C} that recognizes L . In this case, T_L is termed a *characteristic sample* for the algorithm \mathbf{A} . In some cases (e.g., DFAs) there is also a polynomial time algorithm to compute a characteristic sample for \mathbf{A} , given an acceptor $\mathcal{C} \in \mathbb{C}$.

In *active learning* the model of *query learning* [5] assumes the learner communicates with an *oracle* (sometimes called *teacher*) that can answer certain types of queries about the language. The most common type of queries are *membership queries* (is $w \in L$ where L is the unknown language) and *equivalence queries* (is $\llbracket \mathcal{A} \rrbracket = L$ where \mathcal{A} is the current hypothesis for an acceptor recognizing L). Equivalence queries are typically assumed to return a counterexample, i.e. a word in $\llbracket \mathcal{A} \rrbracket \setminus L$ or in $L \setminus \llbracket \mathcal{A} \rrbracket$.

With regard to ω -automata (automata on infinite words) most of the works consider *query learning*. The representations learned so far include: $(L)_{\S}$ [15], a non-polynomial reduction to finite words; families of DFAs (FDFAs) [7,8,6,22]; strongly unambiguous Büchi automata (SUBA) [3]; and deterministic weak parity automata (DWPA) [23]. Among these only the latter is learnable in polynomial time using membership queries and proper equivalence queries.

One of the main obstacles in obtaining a polynomial learning algorithm for ω -regular languages is that they do not in general have a Myhill-Nerode characterization; that is, there is no theorem correlating the states of a minimal automaton of some of the common automata types (Büchi, Parity, Muller, etc.)

to the equivalence classes of the right congruence of the language. The *right congruence relation* for an ω -language L relates two finite words x and y iff there is no infinite suffix z differentiating them, that is $x \sim_L y$ (for $x, y \in \Sigma^*$) iff $\forall z \in \Sigma^\omega. xz \in L \iff yz \in L$. In our quest for finding a polynomial query learning algorithm for a subclass of the ω -regular languages, we have studied subclasses of languages for which such a relation holds [4], and termed them *fully informative*. We use $\mathbb{IB}, \mathbb{IC}, \mathbb{IP}, \mathbb{IM}$ to denote the classes of languages that are fully informative of type Büchi, coBüchi, Parity and Muller, respectively. A language L is said to be fully informative of type \mathbb{X} for $\mathbb{X} \in \{\mathbb{B}, \mathbb{C}, \mathbb{P}, \mathbb{M}\}$ if there exists a deterministic automaton of type \mathbb{X} which is isomorphic to the automaton derived from \sim_L . While a lot of properties about these classes are now known, in particular that they span the entire hierarchy of ω -regular properties [34], a polynomial learning algorithm for them has not been found yet.

In this paper we show that the classes $\mathbb{IB}, \mathbb{IC}, \mathbb{IP}, \mathbb{IM}$ can be identified in the limit using polynomial time and data. We further show that there is a polynomial time algorithm to compute a characteristic sample given an acceptor $\mathcal{C} \in \mathbb{IX}$. A corollary of this result is that the class of languages accepted by \mathbb{DWPAs} (which as mentioned above is polynomially learnable in the query learning setting) also has a polynomial size characteristic sample. On the negative side, we show that the classes $\mathbb{NBA}, \mathbb{NCA}, \mathbb{NPA}, \mathbb{NMA}$ of non-deterministic Büchi, coBüchi, Parity and Muller automata, resp., cannot be identified in the limit using polynomial data.

2 Preliminaries

Automata and Acceptors An *automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_i, \delta \rangle$ consisting of a finite totally ordered alphabet Σ of symbols, a finite set Q of states, an initial state $q_i \in Q$, and a transition function $\delta : Q \times \Sigma \rightarrow 2^Q$. A run of an automaton on a finite word $v = a_1 a_2 \dots a_n$ is a sequence of states q_0, q_1, \dots, q_n such that $q_0 = q_i$, and for each $i \geq 0, q_{i+1} \in \delta(q_i, a_{i+1})$. A run on an infinite word is defined similarly and results in an infinite sequence of states. We say that \mathcal{A} is *deterministic* if $|\delta(q, a)| \leq 1$ and *complete* if $|\delta(q, a)| \geq 1$, for every $q \in Q$ and $a \in \Sigma$. We extend δ to domain $Q \times \Sigma^*$ in the usual manner, and abbreviate $\delta(q, \sigma) = \{q'\}$ as $\delta(q, \sigma) = q'$.

By augmenting an automaton with an acceptance condition α , obtaining a tuple $\langle \Sigma, Q, q_i, \delta, \alpha \rangle$, we get an *acceptor*, a machine that accepts some words and rejects others. An acceptor accepts a word if at least one of the runs on that word is accepting. For finite words the acceptance condition is a set $F \subseteq Q$ and a run on a word v is accepting if it ends in an accepting state, i.e., if $\delta(q_i, v)$ contains an element of F . For infinite words, there are various acceptance conditions in the literature; we consider four: Büchi, coBüchi, parity, and Muller. The Büchi and coBüchi acceptance conditions are also a set $F \subseteq Q$. A run of a Büchi automaton is accepting if it visits F infinitely often. A run of a coBüchi is accepting if it visits F only finitely many times. A parity acceptance condition is a map $\kappa : Q \rightarrow \mathbb{N}$ assigning each state a natural number termed a color (or priority). A run is accepting if the **minimum** color visited infinitely often is **odd**.

A Muller acceptance condition is a set of sets of states $\alpha = \{F_1, F_2, \dots, F_k\}$ for some $k \in \mathbb{N}$ and $F_i \subseteq Q$ for $i \in [1..k]$. A run of a Muller automaton is accepting if the set S of states visited infinitely often in the run is a member of α . We use $\llbracket \mathcal{A} \rrbracket$ to denote the set of words accepted by a given acceptor \mathcal{A} . We use NBA, NPA, NMA, NCA for non-deterministic Büchi, parity, Muller and coBüchi, automata. We use NBA, NPA, NMA and NCA for the classes of languages they recognize. The first three recognize the full class of ω -regular languages while the fourth only a subset of it.

Right congruences An equivalence relation \sim on Σ^* is a *right congruence* if $x \sim y$ implies $xv \sim yv$ for every $x, y, v \in \Sigma^*$. The *index* of \sim , denoted $|\sim|$ is the number of equivalence classes of \sim . Given a language $L \subseteq \Sigma^*$ its *canonical right congruence* \sim_L is defined as follows: $x \sim_L y$ iff $\forall z \in \Sigma^*$ we have $xz \in L \iff yz \in L$. For a word $v \in \Sigma^*$ the notation $[v]$ is used for the equivalence class of \sim in which v resides.

With a right congruence \sim of finite index one can naturally associate an automaton $\mathcal{M}_\sim = \langle \Sigma, Q, q_i, \delta \rangle$ as follows: the set of states Q consists of the equivalence classes of \sim . The initial state q_i is the equivalence class $[\varepsilon]$. The transition function δ is defined by $\delta([u], a) = [ua]$. Similarly, given a complete deterministic automaton $\mathcal{M} = \langle \Sigma, Q, q_i, \delta \rangle$ we can naturally associate with it a right congruence as follows: $x \sim_{\mathcal{M}} y$ iff \mathcal{M} reaches the same state when reading x or y . The Myhill-Nerode Theorem states that a language L is regular iff \sim_L is of finite index. Moreover, if L is accepted by a DFA \mathcal{A} then $\sim_{\mathcal{A}}$ refines \sim_L . Finally, the index of \sim_L gives the size of the minimal complete DFA for L .

For an ω -language $L \subseteq \Sigma^\omega$, the right congruence \sim_L is defined similarly, by quantifying over ω -words. That is, $x \sim_L y$ iff $\forall z \in \Sigma^\omega$ we have $xz \in L \iff yz \in L$. Given a deterministic automaton \mathcal{M} we can define $\sim_{\mathcal{M}}$ exactly as for finite words. However, for ω -regular languages, the relation \sim_L does not suffice to obtain a “Myhill-Nerode” characterization. As an example consider the language $L = (a+b)^*(bba)^\omega$. We have that \sim_L consists of just one equivalence class, since for any $x \in \Sigma^*$ and $w \in \Sigma^\omega$ we have that $xw \in L$ iff w has $(bba)^\omega$ as a suffix. But an ω -acceptor recognizing L obviously needs more than a single state.

The classes \mathbb{IB} , \mathbb{IC} , \mathbb{IP} and \mathbb{IM} A language L is in \mathbb{IB} (resp., \mathbb{IC} , \mathbb{IP} , \mathbb{IM}) if there exists a deterministic Büchi (resp., coBüchi, parity, Muller) acceptor \mathcal{A} such that $L = \llbracket \mathcal{A} \rrbracket$ and there is a 1-to-1 relationship between the states of \mathcal{A} and the equivalence classes of \sim_L : if $x \sim_L y$ then x and y reach the same state q in \mathcal{A} , and an ω -word z is accepted from q iff $xz \in L$ (which holds iff $yz \in L$). These classes are more expressive than one might conjecture, it was shown in [4] that in every class of the infinite Wagner hierarchy [34] there are languages in \mathbb{IM} and \mathbb{IP} . Moreover, in a small experiment reported in [4], among randomly generated Muller automata, the vast majority turned out to be in \mathbb{IM} .

Examples and samples Since we need finite representations of examples, ω -words in our case, we work with ultimately periodic words, that is, words of the form $u(v)^\omega$ where $u \in \Sigma^*$ and $v \in \Sigma^+$. It is known that two regular ω -languages are

equivalent iff they agree on the set of ultimately periodic words, so this choice is not limiting. The example $u(v)^\omega$ is concretely represented by the pair (u, v) of finite strings, and its length is $|u| + |v|$. A *labeled example* is a pair $(u(v)^\omega, l)$, where the label l is either 0 or 1. A *sample* is a finite set of labeled examples such that no example is assigned two different labels. The length of a sample is the sum of the lengths of the examples that appear in it. A sample T and a language L are consistent with each other if and only if for every labeled example $(u(v)^\omega, l) \in T$, $l = 1$ iff $u(v)^\omega \in L$. A sample and an acceptor are consistent with each other if and only if the sample and the language recognized by the acceptor are consistent with each other. The following results give two useful procedures on examples that are computable in polynomial time.

Claim 1. *Given $u_1, u_2 \in \Sigma^*$ and $v_1, v_2 \in \Sigma^+$, if $u_1(v_1)^\omega \neq u_2(v_2)^\omega$ then they differ in at least one of the first ℓ symbols, where $\ell = \max(|u_1|, |u_2|) + |v_1| \cdot |v_2|$.*

Let $\text{suffixes}(u(v)^\omega)$ denote the set of all ω -words that are suffixes of $u(v)^\omega$.

Claim 2. *The set $\text{suffixes}(u(v)^\omega)$ consists of at most $|u| + |v|$ different examples: one of the form $u'(v)^\omega$ for every nonempty suffix u' of u , and one of the form $(v_2v_1)^\omega$ for every division of v into a non-empty prefix and suffix as $v = v_1v_2$.*

Identification in the limit using polynomial time and data We consider the notion of identification in the limit using polynomial time and data. This criterion of learning was introduced by [16], who showed that regular languages of finite strings represented by DFAs are learnable in this sense. We follow a more general definition given by [19]. The definition has two requirements: (1) a learning algorithm **A** that runs in polynomial time on a set of labeled examples and produces a hypothesis consistent with the examples, and (2) that for every language L in the class, there exists a set T_L of labeled examples of size polynomial in a measure of size of L such that on any set of labeled examples containing T_L , the algorithm **A** outputs a hypothesis correct for L . Condition (1) ensures polynomial time, while condition (2) ensures polynomial data. The latter is not a worst-case measure; there could be arbitrarily large finite samples for which **A** outputs an incorrect hypothesis. However, de la Higuera shows that identifiability in the limit with polynomial time and data is closely related to a model of a learner and a helpful teacher introduced by [17].

3 Negative Results

We start with negative results. We show that when the representation at hand is non-deterministic, polynomial identification is not feasible.

Theorem 3. *The class NBA cannot be identified in the limit using polynomial data.*

Proof. The proof follows the idea given in the negative result for learning in the limit NFAs from polynomial data [19]. For any integer $M \geq 2$, let p_1, \dots, p_m be

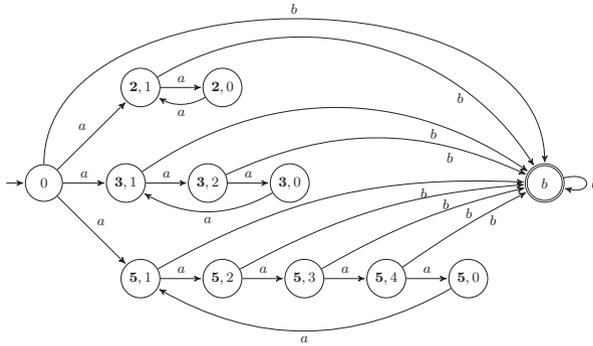


Fig. 1: The NBA \mathcal{B}_M for $M = 5$.

the set of all primes less than or equal to M . For each such M , consider the NBA \mathcal{B}_M over a two letter alphabet $\Sigma = \{a, b\}$ with $p_1 + p_2 + \dots + p_m + 2$ states, where state 0 has a -transitions to state $(\mathbf{p}, 1)$ for each $\mathbf{p} \in \{p_1, p_2, \dots, p_m\}$. State (\mathbf{p}, i) has an a -transition to state $(\mathbf{p}, i \oplus_p 1)$ where \oplus_p is addition modulo p . All states except the states $(\mathbf{p}, 0)$ have a b -transition to state b . The state b has a self-loop on b . The only accepting state is b . The NBA \mathcal{B}_M for $M = 5$ is given in Fig. 1.

The NBA \mathcal{B}_M accepts the set of all words of the form $a^k b^\omega$ such that k is *not* a positive multiple of $\ell = p_1 \cdot p_2 \cdot \dots \cdot p_m$. Note that the size of the shortest ultimately periodic word in $a^* b^\omega \setminus \llbracket \mathcal{B}_M \rrbracket$ is $\ell + 1$, and thus, to distinguish the language $\llbracket \mathcal{B}_M \rrbracket$ from the language $a^* b^\omega$, a word of at least this size must be provided. Since the number of primes not greater than M is $\Theta(M / \log M)$ and since each prime is of size at least 2 the data must be of size at least $2^{\Theta(M / \log M)}$ while the number of states of \mathcal{B}_M is $O(M^2)$. \square

Since NBAs are a special case of non-deterministic parity automata (NPA) and non-deterministic Muller automata (NMA) it follows that these models too cannot be identified in the limit using polynomial data. Note that indeed the NBA in the proof of Theorem 3 can be regarded as an NPA by setting the color of state b to 1 and the color of all other states to 0. Likewise it can be regarded as an NMA by defining the accepting set as $\{\{b\}\}$.

Corollary 1. *The classes NPA and NMA cannot be identified in the limit using polynomial data.*

While NBAs are not a special case of non-deterministic coBüchi automata (NCA) it can be shown that NCA as well cannot be identified in the limit from polynomial data, which is in some sense surprising, since NCAs are not more expressive than DCAs, their deterministic counterpart, and accept a very small subclass of the regular ω -languages.

Theorem 4. *The class NCA cannot be identified in the limit using polynomial data.*

Proof. The proof is almost identical to that of Theorem 3. The only difference is that it considers the automaton \mathcal{C}_M that takes exactly the same form as \mathcal{B}_M from that proof but switching accepting and non-accepting states. Since \mathcal{C}_M clearly accepts the same language as that of \mathcal{B}_M , with the same number of states, the proof continues exactly the same. \square

4 Outline for the positive results

The rest of the paper is devoted to the positive results. To show that a class is identified in the limit using polynomial time and data there are two steps: (i) constructing a sample of words T_L of size polynomial in the given acceptor \mathcal{M} for the language L at hand, the so called, *characteristic sample*, and (ii) providing a learning algorithm that for every given sample T returns an acceptor consistent with that sample, and in addition for any sample T that subsumes T_L returns an acceptor that exactly recognizes L .

Since the construction of the characteristic sample is simpler we start with that. We show that the classes \mathbb{IB} , \mathbb{IC} , \mathbb{IP} and \mathbb{IM} have characteristic samples of size polynomial in the number of states of the acceptor, and that the characteristic sample can be constructed in polynomial time. The definition of an acceptor is composed of two steps: (a) the definition of the automaton and (b) the definition of the acceptance condition. Some words are put in the sample to help retrieving the automaton and some to help retrieving the acceptance condition. We view the characteristic sample as a union of two parts T_{Aut} (for retrieving the automaton) and T_{Acc} (for retrieving the acceptance condition). The learning algorithm first constructs the automaton, then retrieves the acceptance condition.

In Section 5 we discuss the construction of T_{Aut} which is common to all the classes we consider, as they all are isomorphic to the automaton of the right congruence. In Section 6 we show how an algorithm can retrieve the automaton using the labeled words in T_{Aut} . In Section 7 we discuss the construction of T_{Acc} that regards the acceptance condition of the DPA. This part is the most involved one. We first associate with a DPA a canonical forest of its strongly connected components. From this canonical forest we build the T_{Acc} part of the characteristic sample. In Section 8 we show a learning algorithm that can retrieve in polynomial time the acceptance condition of the DPA, from labeled examples in T_{Acc} . This implies that \mathbb{IP} (as well as its special cases \mathbb{IB} and \mathbb{IC}) can be learned in the limit from polynomial time and data. In Section 9 we show that the class \mathbb{IM} can also be learned in the limit from polynomial time and data.

5 The characteristic sample for the automaton

In this section we show how to construct the T_{Aut} part of the sample. We first show that any two states that are distinguishable in the automaton, are distinguishable by words of length polynomial in the number of states.

5.1 Polynomial construction of short distinguishing words

Let M be an acceptor in one of the classes $\mathbb{I}\mathbb{B}$, $\mathbb{I}\mathbb{C}$, $\mathbb{I}\mathbb{P}$ or $\mathbb{I}\mathbb{M}$ with states Q over alphabet Σ . If M is in one of the first three classes we use $\max\{|\Sigma|, |Q|\}$ for its size measure. If $M \in \mathbb{I}\mathbb{M}$ we use $\max\{|\Sigma|, |Q|, m\}$ for its size measure where m is the number of sets in the acceptance condition α . We say that states q_1 and q_2 of M are *distinguishable* if there exists a word $z \in \Sigma^\omega$ that is accepted from one but not the other (and that z is a *distinguishing word*). First we show that any two distinguishable states of M are distinguishable by an ultimately periodic word of size polynomial in M . Then we show how to use these words to construct the T_{Aut} part of the characteristic sample.

Proposition 5. *If two states of a DMA, DPA, DBA or DCA of n states are distinguishable, then they are distinguishable by an ultimately periodic ω -word of length bounded by $n^2 + n^4$.*

Proof. We prove that for a DMA M of n states, if two distinct states q_1 and q_2 are distinguishable, then they are distinguishable by an ultimately periodic ω -word of length bounded by $n^2 + n^4$. Since any DPA, DBA or DCA is equivalent to an isomorphic DMA, the above result holds also for DPAs, DBAs and DCAs.

Because q_1 and q_2 are distinguishable, there exists an ultimately periodic ω -word $x(y)^\omega$ that is accepted from exactly one of the two states. For each nonnegative integer k and $i = 1, 2$, let $q_i(k)$ be the state visited after k symbols of $x(y)^\omega$ have been read, starting with state q_i . Also, let C_i be the set of states visited infinitely often by the sequence $q_i(k)$, which determines the acceptance or rejection of $x(y)^\omega$ from q_i . The sequence of pairs $(q_1(k), q_2(k))$ for $k = 0, 1, \dots$ takes on at most n^2 different values. Let C be the set of pairs visited infinitely often by this sequence. The two projections $\pi_1(C)$ and $\pi_2(C)$ are C_1 and C_2 .

Let ℓ be the minimum value for which $(q_1(k), q_2(k))$ visits only pairs in C for all $k \geq \ell$. Let x' be the prefix of $x(y)^\omega$ consisting of ℓ symbols. By removing symbols between repeated pairs $(q_1(k), q_2(k))$ from x' we obtain a string u of length at most n^2 that reaches the pair $(q_1(\ell), q_2(\ell))$ from $(q_1(0), q_2(0))$. Let m be the minimum value for which $(q_1(k), q_2(k))$ for $\ell \leq k \leq m$ visits all the pairs of C and returns to $(q_1(\ell), q_2(\ell))$, and let y' be the string from symbol ℓ to $m - 1$ of $x(y)^\omega$. Distinguishing a subsequence of pairs that visits each element of C once, we can remove from y' sequences of symbols between repeated pairs that do not include a distinguished pair between them. Thus we obtain a string v of length at most $|C|n^2$, that starts at $(q_1(\ell), q_2(\ell))$, visits all the distinguished pairs and returns to the starting pair. Since $|C| \leq n^2$, the length of $u(v)^\omega$ is at most $n^2 + n^4$. Also, since the set of states visited infinitely often on input $u(v)^\omega$ from q_i is C_i we have that $u(v)^\omega$ is accepted from exactly one of q_1 and q_2 . \square

For DPAs as well as DMAs there is a polynomial time algorithm to determine whether two states are distinguishable and to find a distinguishing ω -word $u(v)^\omega$ if they are. This result relies on a polynomial time algorithm to test the equivalence of two DPAs or two DMAs and return an example $u(v)^\omega$ on which they differ if not [9]. Since DBA and DCA are special cases of a DPA, a polynomial construction of a distinguishing word applies to them as well.

5.2 Constructing the characteristic sample for the automaton

We now show how to construct the T_{Aut} part of the characteristic sample, given an acceptor \mathcal{M} in one of the classes \mathbb{IM} , \mathbb{IP} , \mathbb{IB} or \mathbb{IC} . Let n be the number of states of \mathcal{M} . We may assume that every state of \mathcal{M} is reachable from the initial state q_i . The algorithm constructs a set S of n access strings by breadth-first search in the transition graph of \mathcal{M} such that S is prefix-closed and contains exactly one lexicographically least string of shortest possible length reaching each state of \mathcal{M} from the initial state. Using Proposition 5, the algorithm may also construct a set E of at most n^2 distinguishing experiments that contains for each pair q_1 and q_2 of distinct states of \mathcal{M} , an ω -word $u(v)^\omega$ of length at most $n^2 + n^4$ that is accepted from exactly one of the states q_1 and q_2 .

Part one of the sample, T_{Aut} , consists of all the examples in $(S \cdot E) \cup (S \cdot \Sigma \cdot E)$, labeled to be consistent with \mathcal{M} . There are at most $(1 + |\Sigma|)n^3$ labeled examples in T_{Aut} , each of length bounded by a polynomial in n . This information is enough to allow the polynomial time learning algorithm to reconstruct a transition graph isomorphic to that of \mathcal{M} .

Proposition 6. *Let \mathcal{M}' be any deterministic automaton that is consistent with the sample T_{Aut} . Then \mathcal{M}' has at least n states and if \mathcal{M}' has exactly n states then \mathcal{M}' and \mathcal{M} have isomorphic transition graphs.*

Proof. The states of \mathcal{M}' reached from the initial state by the access strings in S must all be distinct, because for any pair of different strings $s_1, s_2 \in S$, there exists a word $u(v)^\omega \in E$ such that $s_1 \cdot u(v)^\omega$ and $s_2 \cdot u(v)^\omega$ have different labels in T_{Aut} . Thus \mathcal{M}' must have at least n distinct states.

Assume that \mathcal{M}' has exactly n states. Given the state q of \mathcal{M}' reached by some $s \in S$ and a symbol $\sigma \in \Sigma$, the labeled examples $s \cdot \sigma \cdot u(v)^\omega$ in T_{Aut} for all $u(v)^\omega \in E$ uniquely determine which string $s' \in S$ corresponds to the state reached in \mathcal{M}' from q on input symbol σ . Thus the transition graph of \mathcal{M}' is isomorphic to the transition graph of \mathcal{M} . □

6 Learning the automaton

Let L denote the language to be learned, and \mathcal{M} denote an acceptor of n states that is isomorphic to its right congruence automaton and recognizes L . Let the input sample of labeled examples be T . We now describe a learning algorithm **A** that makes use of the information in the given sample T to construct an automaton. If T subsumes T_{Aut} the returned automaton will be isomorphic to the acceptor \mathcal{M} .

From the sample T , the algorithm constructs as follows a set E of strings that serve as experiments used to distinguish states. For each labeled example $(u(v)^\omega, l)$ in T , all of the elements of $\text{suffixes}(u(v)^\omega)$ are placed in E . Thus if the sample T includes T_{Aut} , then for any pair of states of \mathcal{M} the set E includes an experiment that distinguishes them.

Starting with the empty string ε , the algorithm attempts to build up a prefix-closed set S of finite strings that reach different states of \mathcal{M} from the initial state.

Initially, $S_1 = \{\varepsilon\}$. After S_k has been constructed, the algorithm attempts to determine, for each $s \in S_k$ and each symbol $\sigma \in \Sigma$ in the ordering defined on Σ , whether $s \cdot \sigma$ reaches the same state as some string already in S_k or a new state. If for each string s' in S_k , there exists some $u(v)^\omega \in E$ such that the sample T has different labels for $s \cdot \sigma \cdot u(v)^\omega$ and $s' \cdot u(v)^\omega$, then this is evidence that $s \cdot \sigma$ reaches a new state, and S_{k+1} is set to $S_k \cup \{s \cdot \sigma\}$. If no such pair s and σ is found, then the final set S is S_k . Because \mathcal{M} has only n states, this case is reached with $k \leq n$. If the sample T subsumes T_{Aut} then this process will discover exactly the strings reaching all n states of \mathcal{M} used in the construction of T_{Aut} ; otherwise, it may terminate early.

In the second phase, the algorithm uses the strings in S as names for states and constructs a transition function δ' using S and E . For each $s \in S$ and $\sigma \in \Sigma$, we know that there is at least one $s' \in S$ such that there is no $u(v)^\omega \in E$ for which $s \cdot \sigma \cdot u(v)^\omega$ and $s' \cdot u(v)^\omega$ have different labels in T (possibly because one or more of these examples are not in T at all.) The algorithm selects one such s' and defines $\delta'(s, \sigma) = s'$. If the strings in S actually reach all the states of \mathcal{M} and the choice of s' is unique in each case, then δ' will be isomorphic to the transition function of \mathcal{M} . This will be the case if the sample T includes T_{Aut} because then among the elements of E will be experiments that distinguish any pair of states of \mathcal{M} ; otherwise, δ' may not be correct.

7 Characteristic sample for a DPA

The construction of T_{Acc} , the part of the characteristic sample used for retrieving the accepting condition of a DPA, builds on the construction of a forest of SCCs associated with a given DPA, which we term the *canonical forest*. Its properties and its construction are described next.

7.1 Constructing the canonical forest of a DPA

We start with some definition and simple claims. Let $\mathcal{P} = (\Sigma, Q, q_i, \delta, \kappa)$ be a deterministic parity acceptor (DPA). A set of states $C \subseteq Q$ is a *strongly connected component* (SCC) if and only if C is nonempty and for every $q_1, q_2 \in C$, there exists a nonempty string $v \in \Sigma^+$ such that $\delta(q_1, v) = q_2$ and for all $u \preceq v$, $\delta(q_1, u) \in C$. Note that an SCC need not be maximal, and that a singleton $\{q\}$ is an SCC if and only if the state q has a self-loop, that is, $\delta(q, \sigma) = q$ for some $\sigma \in \Sigma$. For any ω -word w , the set C of states visited infinitely often in the run of \mathcal{P} on input w is an SCC of \mathcal{P} .

Claim 7. *If C_1 and C_2 are SCCs of \mathcal{P} and $C_1 \cap C_2 \neq \emptyset$, then $C_1 \cup C_2$ is also an SCC of \mathcal{P} .*

If \mathcal{P} is a DPA and $R \subseteq Q$ is any set of states, define $SCCs(R)$ to be the set of all C such that $C \subseteq R$ and C is an SCC of \mathcal{P} . Also define $maxSCCs(R)$ to be the maximal elements of $SCCs(R)$ with respect to the subset ordering.

Claim 8. *If \mathcal{P} is a DPA and $R \subseteq Q$ is any set of states, then the elements of $\text{maxSCCs}(R)$ are pairwise disjoint, and every set $C \in \text{SCCs}(R)$ is a subset of exactly one element of $\text{maxSCCs}(R)$.*

If \mathcal{P} is a DPA, we extend its coloring function κ to any nonempty set R of states by $\kappa(R) = \min\{\kappa(q) \mid q \in R\}$. We define the *parity* of R to be 1 if $\kappa(R)$ is odd, and 0 otherwise. For an ω -word w , if the SCC C is the set of states visited infinitely often in the run of \mathcal{P} on w , then w is accepted by P iff the parity of C is 1. Note that the union of two sets of parity b is also of parity b . For any set of states $R \subseteq Q$, we define $\text{minStates}(R)$ to be the set of states $q \in R$ such that $\kappa(q) = \kappa(R)$, that is, the states of R that are assigned the minimum color among all states of R .

The Canonical Forest Using these definitions we can show that there exists a forest associated with a DPA that has the following interesting properties. We provide an example for a canonical forest for a given DPA at the end of the current subsection.

Theorem 9. *Let $\mathcal{P} = (\Sigma, Q, q_0, \delta, \kappa)$ be a DPA. There exists a canonical forest $F^*(\mathcal{P})$ that is unique up to isomorphism and has the following properties.*

1. *There are at most $|Q|$ nodes in $F^*(\mathcal{P})$, each one a distinct SCC of \mathcal{P} .*
2. *The root nodes of $F^*(\mathcal{P})$ are the elements of $\text{maxSCCs}(Q)$.*
3. *The children of a node C of parity b are the maximal SCCs $C' \subseteq C$ of parity $1 - b$.*
4. *The children of a node C are pairwise disjoint and their union is a proper subset of C .*
5. *For any SCC D of \mathcal{P} , there is a unique node C in $F^*(\mathcal{P})$ such that $D \subseteq C$ and D is not a subset of any of the children of C , and C and D have the same parity.*

Proof. The root nodes of $F^*(\mathcal{P})$ are the elements of $\text{maxSCCs}(Q)$ and are SCCs that are pairwise disjoint, by Claim 8. Let C be one of them, and assume its parity is b . Let T be the set of SCCs that are subsets of C and of parity $1 - b$. If $T = \emptyset$ then C has no children and is a leaf of $F^*(\mathcal{P})$. Otherwise, the children of C are the maximal elements of T with respect to the subset ordering. The children of C must be pairwise disjoint because if they share a state, then their union is an SCC contained in C of parity $1 - b$ and is a proper superset of at least one of them, violating maximality. No child of C can contain an element of $\text{minStates}(C)$ because otherwise the parity of the child would be b . Thus the union of the children of C must be a proper subset of C . These conditions imply that there are at most $|Q|$ nodes in the forest, and that it is unique up to isomorphism.

Let D be any SCC of \mathcal{P} . Then $D \in \text{SCCs}(Q)$, so by Claim 8, because the roots of $F^*(\mathcal{P})$ are the elements of $\text{maxSCCs}(Q)$, there is a unique root node C_0 such that $D \subseteq C_0$. Suppose the parity of C_0 is b . If D is not a subset of any of the children of C_0 , then it cannot have parity $1 - b$, so the choice $C = C_0$

satisfies the required condition. If, however, D is a subset of some child C_1 of C_0 , then because the children of C_0 are pairwise disjoint, C_1 is the only child of C_0 that contains D . Again, if D is not a subset of any of the children of C_1 then D and C_1 must have the same parity, and the choice $C = C_1$ satisfies the condition. Otherwise, we continue down the tree rooted at C_0 until a node C is found that satisfies the condition. Note that if we arrive at a leaf C_k , then D is not a subset of any of the children of C_k (there are none) and D must have the same parity as C_k because otherwise C_k would have at least one child. \square

The Canonical Coloring The canonical forest $F^*(\mathcal{P})$ allows us to define a canonical coloring κ^* for \mathcal{P} , as follows. The states in $(Q \setminus \bigcup \text{maxSCCs}(Q))$ are not contained in any SCC of \mathcal{P} and do not affect the acceptance or rejection of any ω -word. For definiteness, we assign them $\kappa^*(q) = 0$. For each node C of $F^*(\mathcal{P})$, we define $\Delta(C)$ to be the set of states of C that are not contained in the union of the children of C . For a root node C of parity b , we define $\kappa^*(q) = b$ for all $q \in \Delta(C)$. Let C be an arbitrary node of $F^*(\mathcal{P})$. If the states of $\Delta(C)$ have been assigned color k by κ^* and D is a child of C , then the states of $\Delta(D)$ are assigned color $k + 1$ by κ^* . We observe that if $q_1 \in \Delta(C)$ and q_2 is in a child of C , then $\kappa^*(q_1) < \kappa^*(q_2)$, and $\kappa^*(q_1)$ is of the same parity as C .

Theorem 10. *Let $\mathcal{P} = (\Sigma, Q, q_0, \delta, \kappa)$ be a DPA, and \mathcal{P}' be \mathcal{P} with the canonical coloring κ^* for \mathcal{P} in place of κ . Then \mathcal{P} and \mathcal{P}' recognize the same ω -language.*

Proof. Let w be an ω -word and let D be the SCC consisting of the states visited infinitely often in the run of \mathcal{P} (and also of \mathcal{P}') on input w . Let C be the unique node of $F^*(\mathcal{P})$ such that D is a subset of C and is not a subset of any of the children of C . Thus D contains at least one $q \in \Delta(C)$. In \mathcal{P} the parity of D is the same as the parity of C , which is the same as the parity of $\kappa^*(q)$, which is equal to the parity of D in \mathcal{P}' . Thus either both \mathcal{P} and \mathcal{P}' accept w or both reject w . \square

Computing the Canonical Forest We now show that, given a DPA $\mathcal{P} = (\Sigma, Q, q_0, \delta, \kappa)$, we can compute the canonical forest of \mathcal{P} in polynomial time. We first define a (possibly non-canonical) forest $F_\kappa(\mathcal{P})$ using the given coloring κ . The root nodes are the elements of $\text{maxSCCs}(Q)$, the set of all maximal SCCs of \mathcal{P} . Once we have defined a node C of the forest, the children are the elements of the set $\text{maxSCCs}(C \setminus \text{minStates}(C))$, that is, the maximal SCCs contained in C with the set of states of minimum color removed. If this set is empty, the node has no children and is a leaf. Note that in contrast to the case of the canonical forest, in $F_\kappa(\mathcal{P})$ the children of a node are not constrained to be of parity opposite to that of the parent.

By construction each node in the forest $F_\kappa(\mathcal{P})$ is an SCC of \mathcal{P} . If D is a descendant of C in the forest, then D is a proper subset of C , and $\kappa(C) < \kappa(D)$. Because the roots are pairwise disjoint and the children of any node are pairwise disjoint, the sets $\text{minStates}(C)$ for nodes C in the forest are pairwise disjoint and

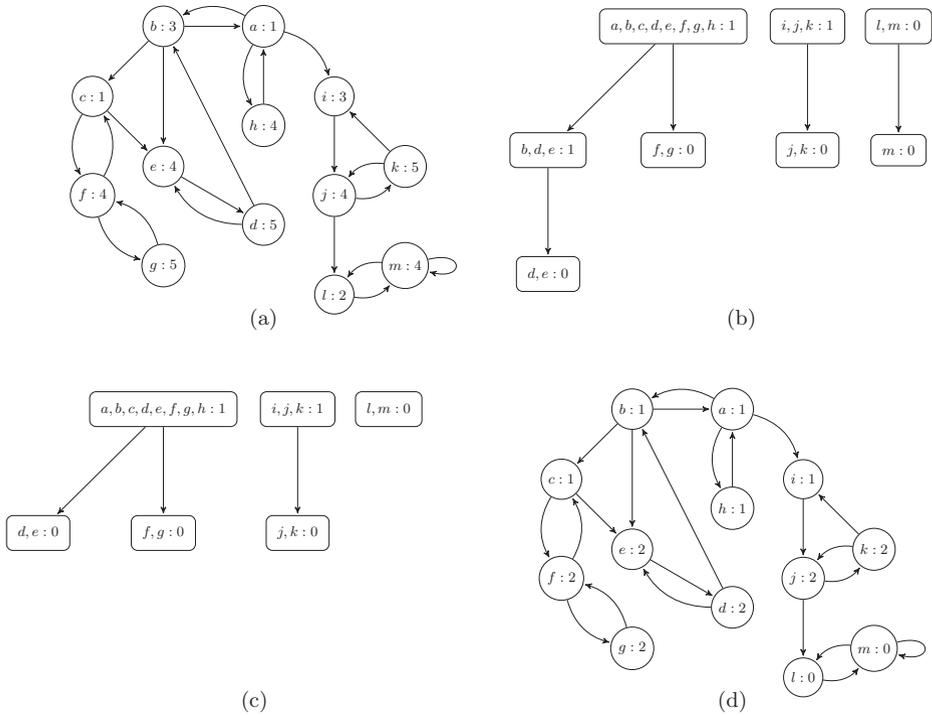


Fig. 2: (a) Transition graph of DPA \mathcal{P} with states colored by κ . (b) Non-canonical forest $F_\kappa(\mathcal{P})$, with parities of nodes. (c) Canonical forest $F^*(\mathcal{P})$, with parities of nodes. (d) Transition graph of \mathcal{P} with the canonical coloring κ^* .

nonempty, so there are at most $|Q|$ nodes. Because a linear time algorithm for computing strongly connected components can be used to compute the children of a node, the forest $F_\kappa(\mathcal{P})$ may be computed in polynomial time in the size of the given DPA \mathcal{P} .

To obtain the canonical forest $F^*(\mathcal{P})$ from the possibly non-canonical forest $F_\kappa(\mathcal{P})$, we may repeatedly merge pairs of adjacent nodes of the same parity until every pair of adjacent nodes are of different parity. That is, if C is a node of parity b and D is a child of C of parity b , then $D \subseteq C$, and we merge D into C by deleting D and making all the children of D direct children of C . Repeating this operation until there are no parent/child pairs of equal parity yields the canonical forest $F^*(\mathcal{P})$. This computation can be done in polynomial time.

Note that to obtain a canonical forest for a given DBA (resp., DCA) we can simply first color states in F by 1 (resp. 0) and in $Q \setminus F$ by 2 (resp., 1) and then compute the canonical forest for the resulting DPA. In both cases the canonical forest will be of depth at most two, since in DBA an accepting SCC cannot be subsumed by a rejecting SCC (and vice versa in DCA).

An Example Figure 2(a) shows the transition graph of an example DPA \mathcal{P} with states a through m , labeled by the colors assigned by κ . There is a directed edge from state q_1 to state q_2 if there exists a symbol $\sigma \in \Sigma$ such that $\delta(q_1, \sigma) = q_2$.

Figure 2(b) shows the non-canonical SCC forest $F_\kappa(\mathcal{P})$ of \mathcal{P} , with the nodes labeled by their parities. Figure 2(c) shows the canonical SCC forest $F^*(\mathcal{P})$ of \mathcal{P} , with the nodes labeled by their parities. Figure 2(d) shows the transition graph of \mathcal{P} re-colored using the canonical coloring κ^* .

7.2 Constructing the characteristic sample for a DPA

We can now construct T_{Acc} , the second part of the characteristic sample for a DPA \mathcal{P} . The sample T_{Acc} consists of one example $u(v)^\omega$ for each node C of the canonical forest $F^*(\mathcal{P})$, where u is a string that reaches a state q in C from the initial state q_0 , and v is a nonempty string that, starting from q , visits every state of C and no state outside of C and returns to q . The length of the example $u(v)^\omega$ can be taken to be bounded by $n + n^2$. The example $u(v)^\omega$ is labeled 1 if it is accepted by \mathcal{P} and otherwise is labeled 0. Then T_{Acc} contains at most n labeled examples, each of length polynomial in n . The final characteristic sample for $L = \llbracket \mathcal{P} \rrbracket$ is $T_L = T_{Aut} \cup T_{Acc}$. The sample T_L contains $O(|\Sigma|n^3)$ labeled examples, each of length at most $O(n^4)$, which is polynomial in $size(L)$.

8 The learning algorithm for a DPA

We can now describe the learning algorithm **A** that makes use of the information in T_L . Similar to Gold’s construction, the algorithm optimistically assumes that the sample includes a characteristic sample, and if that assumption fails to produce an acceptor consistent with the sample, the algorithm defaults to producing a table-lookup acceptor to ensure that its hypothesis is consistent with the sample. The algorithm we describe is sufficient to establish the theoretical results, but for practical applications much more effort should be expended to find good heuristic choices to avoid defaulting too easily.

Let L denote the language to be learned, and \mathcal{P} denote a DPA of n states that is isomorphic to its right congruence automaton and recognizes L . The first and second phases of the algorithm are as described in Section 6: in the first phase the algorithm builds the set S of states of the automaton, and in the second step it builds the transition relation δ' . In the third phase, the acceptance (namely the coloring) is determined. In this phase, the algorithm may default to returning the table-lookup DPA for T . We first explain the construction of the table-lookup DPA then describe the third phase.

A table-lookup DPA A table-lookup DPA for a given sample T is constructed by finding the shortest prefix of each example $u(v)^\omega$ in T that distinguishes it from all other examples in T and placing these prefixes in a trie-like structure. At each leaf of the trie is a structure accepting (or rejecting, depending on the label of the example) the appropriate suffix of the unique example that arrives at that leaf. By Claim 1, this DPA

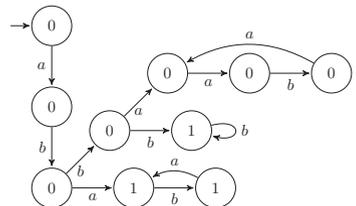


Fig. 3: Table-lookup DPA for $T = \{(a(b)^\omega, 1), ((ab)^\omega, 1), (ab(baa)^\omega, 0)\}$.

can be constructed in time polynomial in the length of the sample T . Note that this construction is easily modified to give a DBA, DCA or DMA instead of a DPA. As an example, for the sample $T = \{(a(b)^\omega, 1), ((ab)^\omega, 1), (ab(baa)^\omega, 0)\}$, the corresponding prefixes are $abbb$, aba , and $abba$, and the table-lookup DPA for T is shown in Figure 3, with states labeled by colors 0 and 1.

Determining the coloring In the third phase, the algorithm attempts to define a coloring of the states in S . The algorithm constructs the set Z of all subsets C of S such that for some labeled example $(u(v)^\omega, l)$ in T , the subset C is the set of elements of S that are visited infinitely often in the run on input $u(v)^\omega$ starting at ε using the transition function δ' . If in this process two examples with different labels are found to yield the same set C , the learning algorithm defaults to the table-lookup DPA for T . Otherwise, each set C in Z is associated with the label of the example(s) that yield C . The set Z is partially ordered by the subset relation. The learning algorithm then attempts to construct a forest F' with nodes that are elements of Z , corresponding to the canonical forest of \mathcal{P} . Initially, F' contains as roots all the maximal elements of Z . If these are not pairwise disjoint, it defaults to the table-lookup DPA for T . Otherwise, for each unprocessed element C in F' , it computes the set of all $D \in Z$ such that $D \subseteq C$, D has the opposite label to C , and D is maximal with these properties, and makes D a child of C . When all the children of a node C have been determined, the algorithm checks two conditions: (1) that the children of C are pairwise disjoint, and (2) there is at least one $s \in C$ that is not in any child of C . If either of these conditions fail, then it defaults to the table-lookup DPA for T . If both conditions are satisfied, then the node C is marked as processed. When there are no more unprocessed nodes, the construction of F' is complete. Note that F' can have at most n nodes, because S has at most n elements.

When the construction of F' completes, for each node C in F' let $\Delta(C)$ denote the elements of C that do not appear in any of its children. Then the learning algorithm assigns colors to the elements of S starting from the roots of F' , as follows. If C is a root with label l , then $\kappa'(s) = l$ for all $s \in \Delta(C)$. If the elements of $\Delta(C)$ have been assigned color k and D is a child of C , then $\kappa'(s) = k + 1$ for all $s \in \Delta(D)$. When this process is complete, any uncolored strings s are assigned $\kappa'(s) = 0$. If the resulting DPA \mathcal{P}' is consistent with the sample T , the learning algorithm outputs \mathcal{P}' and halts. If the sample T includes both T_{Aut} (to specify the automaton) and T_{Acc} (to specify the coloring), then F' will be isomorphic to the canonical forest $F^*(\mathcal{P})$ and κ' will correspond to the canonical coloring κ^* , and \mathcal{P}' will recognize the target language L .

If the process described above does not result in a DPA that is consistent with the sample T , then the algorithm defaults to constructing the table-lookup DPA for T .

The learning algorithm also works for the classes \mathbb{IB} and \mathbb{IC} : In the case of \mathbb{IB} and \mathbb{IC} we need to define a set F rather than a coloring κ . After constructing the forest, the set F is determined to contain the states in the root nodes that are not in the leaves. Thus we have the following.

Theorem 11. *The classes \mathbb{IB} , \mathbb{IC} and \mathbb{IP} are identifiable in the limit using polynomial time and data. Moreover, characteristic samples can be computed in polynomial time.*

A corollary of Theorem 11 is that the class of languages recognized by deterministic weak parity acceptors (\mathbb{DWPA}) which was shown to be polynomially learnable using membership and equivalence queries in [24] is identified in the limit using polynomial time and data. This class (which is equivalent to the intersection of classes $\mathbb{DBA} \cap \mathbb{DCA}$) was shown to be a subset of \mathbb{IM} in [30], and to be a subset of \mathbb{IP} in [4].

Corollary 2. *The class \mathbb{DWPA} is identifiable in the limit using polynomial time and data. Moreover, characteristic samples can be computed in polynomial time.*

9 The sample T_{Acc} and the learning algorithm for a DMA

The above results can be extended to the class \mathbb{IM} . Recall that we define the size measure for a DMA to be $\max\{|\Sigma|, |Q|, m\}$, where m is the number of sets in the acceptance condition. For the characteristic sample T_L , T_{Aut} remains the same, but T_{Acc} contains for each accepting set C , an example $u(v)^\omega$ for which C is the set of states visited infinitely often. In the learning algorithm, the construction of the transition function remains the same. Instead of attempting to construct a coloring function, the learning algorithm finds for each labeled example $(u(v)^\omega, 1) \in T$, the set C of states s that are visited infinitely often on input $u(v)^\omega$ starting from ε and using the transition function δ' , and adds C to the acceptance condition. If the construction does not result in a DMA consistent with T , then it defaults to producing a table-lookup DMA for T . Because in addition, as stated in Section 5.1, a characteristic samples can be computed in polynomial time, we have the following.

Theorem 12. *The class \mathbb{IM} is identifiable in the limit using polynomial time and data. Moreover, a characteristic sample can be computed in polynomial time.*

10 Discussion

We have shown that the non-deterministic classes of ω -automata \mathbb{NBA} , \mathbb{NPA} , \mathbb{NMA} and \mathbb{NCA} cannot be identified in the limit using polynomial data. A negative result regarding query learning of the first three classes was recently obtained in [3]. That result makes a plausible assumption of cryptographic hardness, which is not required here. On the positive side we have shown that the classes \mathbb{IB} , \mathbb{IC} , \mathbb{IP} and \mathbb{IM} can be identified in the limit using polynomial time and data. And moreover, a characteristic sample can be constructed in polynomial time. The construction builds on the definition of a canonical forest for a DPA which may be of use in other contexts as well. The question whether the deterministic classes \mathbb{DBA} , \mathbb{DPA} , \mathbb{DMA} and \mathbb{DCA} can be polynomially learned in the limit remains open.

References

1. Aarts, F., Vaandrager, F.: Learning I/O automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory: 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. pp. 71–85. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
2. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 4–16 (2002)
3. Angluin, D., Antonopoulos, T., Fisman, D.: Strongly unambiguous Büchi automata are polynomially predictable with membership queries. In: 28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain. pp. 8:1–8:17 (2020)
4. Angluin, D., Fisman, D.: Regular omega-languages with an informative right congruence. In: GandALF. EPTCS, vol. 277, pp. 265–279 (2018)
5. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987)
6. Angluin, D., Boker, U., Fisman, D.: Families of DFAs as acceptors of omega-regular languages. In: 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland. pp. 11:1–11:14 (2016)
7. Angluin, D., Fisman, D.: Learning regular omega languages. In: Algorithmic Learning Theory - 25th International Conference, ALT 2014, Bled, Slovenia, October 8-10, 2014. Proceedings. pp. 125–139 (2014)
8. Angluin, D., Fisman, D.: Learning regular omega languages. *Theor. Comput. Sci.* **650**, 57–72 (2016)
9. Angluin, D., Fisman, D.: Polynomial time algorithms for inclusion and equivalence of deterministic omega acceptors. In: arXiv:2002.03191v2, cs.FL (2020)
10. Chalupar, G., Peherstorfer, S., Poll, E., de Ruiter, J.: Automated reverse engineering using Lego®. In: 8th USENIX Workshop on Offensive Technologies (WOOT 14). USENIX Association, San Diego, CA (Aug 2014)
11. Chapman, M., Chockler, H., Kesseli, P., Kroening, D., Strichman, O., Tautschnig, M.: Learning the language of error. In: Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings. pp. 114–130 (2015)
12. Cho, C.Y., Babic, D., Shin, E.C.R., Song, D.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010. pp. 426–439 (2010)
13. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 331–346. TACAS '03, Springer-Verlag, Berlin, Heidelberg (2003)
14. Drews, D., D'Antoni, L.: Learning symbolic automata. In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. pp. 173–189 (2017)
15. Farzan, A., Chen, Y.F., Clarke, E., Tsay, Y.K., Wang, B.Y.: Extending automated compositional verification to the full class of omega-regular languages. In: TACAS. pp. 2–17 (2008)

16. Gold, E.M.: Complexity of automaton identification from given data. *Information and Control* **37**(3), 302–320 (1978)
17. Goldman, S.A., Mathias, H.D.: Teaching a smarter learner. *J. Comput. Syst. Sci.* **52**(2), 255–267 (1996)
18. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.* **138**(3), 21–36 (2005)
19. de la Higuera, C.: Characteristic sets for polynomial grammatical inference. *Machine Learning* **27**(2), 125–138 (1997)
20. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings.* pp. 251–266 (2012)
21. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. *J. ACM* **47**(2), 312–360 (2000)
22. Li, Y., Chen, Y., Zhang, L., Liu, D.: A novel learning algorithm for büchi automata based on family of dfas and classification trees. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I.* pp. 208–226 (2017)
23. Maler, O., Pnueli, A.: On the learnability of infinitary regular sets. *Inf. Comput.* **118**(2), 316–326 (1995)
24. Maler, O., Pnueli, A.: On the learnability of infinitary regular sets. In: *Proceedings of the Fourth Annual Workshop on Computational Learning Theory, COLT 1991, Santa Cruz, California, USA, August 5-7, 1991.* pp. 128–136 (1991)
25. Manevich, R., Shoham, S.: Inferring program extensions from traces. In: *ICGI. Proceedings of Machine Learning Research*, vol. 93, pp. 139–154. PMLR (2018)
26. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: *HLDVT.* pp. 95–100. IEEE Computer Society (2004)
27. Nam, W., Alur, R.: Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In: *ATVA. Lecture Notes in Computer Science*, vol. 4218, pp. 170–185. Springer (2006)
28. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: *FORTE.* pp. 225–240 (1999)
29. Schuts, M., Hooman, J., Vaandrager, F.W.: Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In: *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings.* pp. 311–325 (2016)
30. Staiger, L.: Finite-state omega-languages. *J. Comput. Syst. Sci.* **27**(3), 434–448 (1983)
31. Vaandrager, F.: Model learning. *Commun. ACM* **60**(2), 86–95 (2017)
32. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using language inference to verify omega-regular properties. In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings.* pp. 45–60 (2005)
33. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: *Banff Higher Order Workshop. Lecture Notes in Computer Science*, vol. 1043, pp. 238–266. Springer (1995)

34. Wagner, K.W.: A hierarchy of regular sequence sets. In: 4th Symposium on Mathematical Foundations of Computer (MFCS). pp. 445–449 (1975)

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

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



SV-COMP 2020



Advances in Automatic Software Verification: SV-COMP 2020

Dirk Beyer 

LMU Munich, Germany



Abstract. This report describes the 2020 Competition on Software Verification (SV-COMP), the 9th edition of a series of comparative evaluations of fully automatic software verifiers for C and Java programs. The competition provides a snapshot of the current state of the art in the area, and has a strong focus on replicability of its results. The competition was based on 11 052 verification tasks for C programs and 416 verification tasks for Java programs. Each verification task consisted of a program and a property (reachability, memory safety, overflows, termination). SV-COMP 2020 had 28 participating verification systems from 11 countries.

Keywords: Formal Verification · Program Analysis · Competition

1 Introduction

The Competition on Software Verification (SV-COMP) serves as the showcase of the state of the art in the area of automatic software verification. SV-COMP 2020 is the 9th edition of the competition and presents an overview of the currently achieved results by tool implementations that are based on the most recent ideas, concepts, and algorithms for fully automatic verification. This competition report describes the (updated) rules and definitions, presents the competition results, and discusses some interesting facts about the execution of the competition experiments. The competition measures its own success by evaluating whether the objectives of the competition were achieved. To the objectives discussed earlier (1-4 [14]) we add two further objectives that deserve mentioning (5-6):

1. provide an overview of the state of the art in software-verification technology and increase visibility of the most recent software verifiers,
2. establish a repository of software-verification tasks that is publicly available for free use as standard benchmark suite for evaluating verification software,
3. establish standards that make it possible to compare different verification tools, including a property language and formats for the results,
4. accelerate the transfer of new verification technology to industrial practice by identifying the strengths of the various verifiers on a diverse set of tasks,
5. educate PhD students and others on performing replicable benchmarking, packaging tools, and running robust and accurate research experiments, and
6. provide research teams that do not have sufficient computing resources with the opportunity to obtain experimental results on large benchmark sets.

© The Author(s) 2020

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12079, pp. 347–367, 2020.

https://doi.org/10.1007/978-3-030-45237-7_21

We now discuss the outcome of SV-COMP 2020 with respect to these objectives: (1) There were 28 participating software systems from 11 countries, using many different technologies (cf. Table 6). SV-COMP is considered an important event in the verification community. (2) The sv-benchmarks repository is considered one of the largest and most diverse collections of verification tasks in C and Java. The community dedicates a lot of maintenance effort, as the issue tracker¹ and the pull requests² on GitHub show. (3) SV-COMP has established a format for defining verification tasks, a standard specification language, and a set of functions to express non-deterministic values. Verification results are validated using verification witnesses and six different validators. (4) We received positive feedback from industry, reporting that it is helpful to look up the newest and best available verification tools, regarding the categories of interest. There are several participating systems from industry since 2017. (5) Participating in SV-COMP is also a challenge because the entry requirements are strict: the tools have to be packaged such that all necessary non-standard components are contained, the tools need to provide meaningful log output, the tool parameters have to be specified in the BENCHEXEC benchmark-definition format, and a tool-info module needs to be implemented. All experiments are required to be fully replicable. It is a motivating experience to observe the learning of first-time participants. (6) Running large-scale performance experiments requires an infrastructure with considerable computing resources — which are not necessarily available to all tool developers. Through this competition and the preruns, the participants get the opportunity to repeatedly run experiments on the full benchmark set of verification tasks of the competition. The preruns and final run sum up to over one million verification runs and ten million witness-validation runs.

Related Competitions. It is well-understood that competitions are an important evaluation method, and there are many other competitions in the field of formal methods. The TOOLympics³ [7] event in 2019 (part of the 25-years-of-TACAS celebration) presented 16 competitions in the area. Most closely related are the competitions RERS⁴ [45] and VerifyThis⁵ [46]. While SV-COMP⁶ performs replicable experiments in a *controlled* environment (dedicated resources, resource limits), the RERS Challenges give more room for exploring combinations of interactive with automatic approaches without limits on the resources, and the VerifyThis Competition focuses on evaluating approaches and ideas rather than on *fully automatic* verification.

Large benchmark collections are extremely important to make approaches comparable and to agree on what constitutes interesting problems to solve. There are other large benchmark collections as well (e.g., by SPEC⁷), but the

¹ <https://github.com/sosy-lab/sv-benchmarks/issues>

² <https://github.com/sosy-lab/sv-benchmarks/pulls>

³ <https://tacas.info/toolympics.php>

⁴ <http://rers-challenge.org>

⁵ <http://etaps2016.verifythis.org>

⁶ <https://sv-comp.sosy-lab.org>

⁷ <https://www.spec.org>

`sv-benchmarks` suite⁸ is (a) free of charge, and (b) tailored to the state of the art in software verification. Benchmark repositories of various competitions and challenges also contribute to each other. For example, the `sv-benchmarks` suite contains programs that were originally used in RERS⁹, in termCOMP¹⁰, and in VerifyThis¹¹. There is a flow of benchmarks in the other direction as well: The competition SMT-COMP [32] uses SMT formulas that were generated from programs of the `sv-benchmarks` collection. For example, the k -induction engine of CPACHECKER was used to generate more than 1000 SMT formulas for the quantifier-free theory of arrays and bit-vectors (QF_ABV)¹².

2 Organization, Definitions, Formats, and Rules

Procedure. SV-COMP 2020's overall organization did not change in comparison to the earlier editions [8, 9, 10, 11, 12, 13, 14]. SV-COMP is an open competition, where all verification tasks are known before the submission of the participating verifiers, which is necessary due to the complexity of the C language. During the *benchmark submission* phase, new verification tasks were collected, classified, and added to the existing benchmark suite (i.e., SV-COMP uses an accumulating benchmark suite), during the *training* phase, the teams inspected the verification tasks and trained their verifiers (also, the verification tasks received fixes and quality improvement), and during the *evaluation* phase, verification runs were performed with all competition candidates, and the system descriptions and archives were reviewed by the competition jury. The participants received the results of their verifier directly via e-mail, and after a few days of inspection, the results were publicly announced on the competition web site. The *Competition Jury* consisted again of the chair and one member of each participating team. Team representatives of the jury are listed in Table 5.

Qualification and License Requirements. As a new feature in SV-COMP 2020, a rule was introduced that allows the organizer to reuse systems that participated in previous years, and to enter new systems, provided that the developers were given the chance to contribute a submission themselves (both options were not used this time). Starting 2018, SV-COMP required that the verifier must be publicly available for download and has a license that

- (i) allows replication and evaluation by anybody (including results publication),
- (ii) does not restrict the usage of the verifier output (log files, witnesses), and
- (iii) allows any kind of (re-)distribution of the unmodified verifier archive.

⁸ <https://github.com/sosy-lab/sv-benchmarks>

⁹ <https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/eca-rers2012/README.txt>

¹⁰ <https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/termination-restricted-15/README.txt>

¹¹ <https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/verifythis/README.txt>

¹² https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks-inc/QF_ABV/tree/master/20190307-CPAchecker_kInduction-SoSy_Lab

```

1 format_version: '1.0'
2
3 # old file name: floppy_true-unreach-call_true-valid-memsafety.i.cil.c
4 input_files: 'floppy.i.cil-3.c'
5
6 properties:
7   - property_file: ../properties/unreach-call.prp
8     expected_verdict: true
9   - property_file: ../properties/valid-memsafety.prp
10     expected_verdict: false
11     subproperty: valid-memtrack

```

Fig. 1: Example task definition for program `floppy.i.cil-3.c`

Validation of Results. The validation of the results based on verification witnesses [19, 20] was done as in previous years (2017–2019), mandatory for *both* answers TRUE or FALSE. A few categories were excluded from validation if the validators did not sufficiently support a certain kind of program or property. Two new validators participated in SV-COMP 2020: NITWIT [66] and METAVAL [25].

Verification Tasks — Explicit Task-Definition Files. The notion of verification tasks did not change and we refer to previous reports for more details [10, 13]. We developed a new format for task definitions that was used for the Java category already in SV-COMP 2019. Technically, we need a verification task (a pair of a program and a specification to verify) to feed as input to the verifier, and an expected result against which we check the answer that the verifier returns. Previously, the above-mentioned three components were specified in the file name of the program; now all the information is stored in an extra file that contains a structured definition of the verification tasks for a program. For each program, the repository contains the program file and a task-definition file. Consider an example program that is available under the name `floppy.i.cil-3.c`: This program comes now with its task-definition file `floppy.i.cil-3.yml`. Figure 1 shows this task definition. The new format was used in SV-COMP 2019 for the Java category [14] and in the competition on software testing, Test-Comp 2019 [15].

The task definition uses the YAML format as underlying structured data format. It contains a version id of the format (line 1) and can contain comments (line 3). The field `input_files` specifies the input program (example: `'floppy.i.cil-3.c'`), which is either one file or a list of files. The field `properties` lists all properties of the specification for this program. Each property has a field `property_file` that specifies the property file (example: `../properties/unreach-call.prp`) and a field `expected_verdict` that specifies the expected result (example: `true`).

Categories, Properties, Scoring Schema, and Ranking. The categories are listed in Tables 7 and 8 and described in detail on the competition web site.¹³ Figure 2 shows the category composition. For the definition of the properties and the property format, we refer to the 2015 competition report [11]. All specifications are available in the directory `c/properties/` of the benchmark

¹³ <https://sv-comp.sosy-lab.org/2020/benchmarks.php>

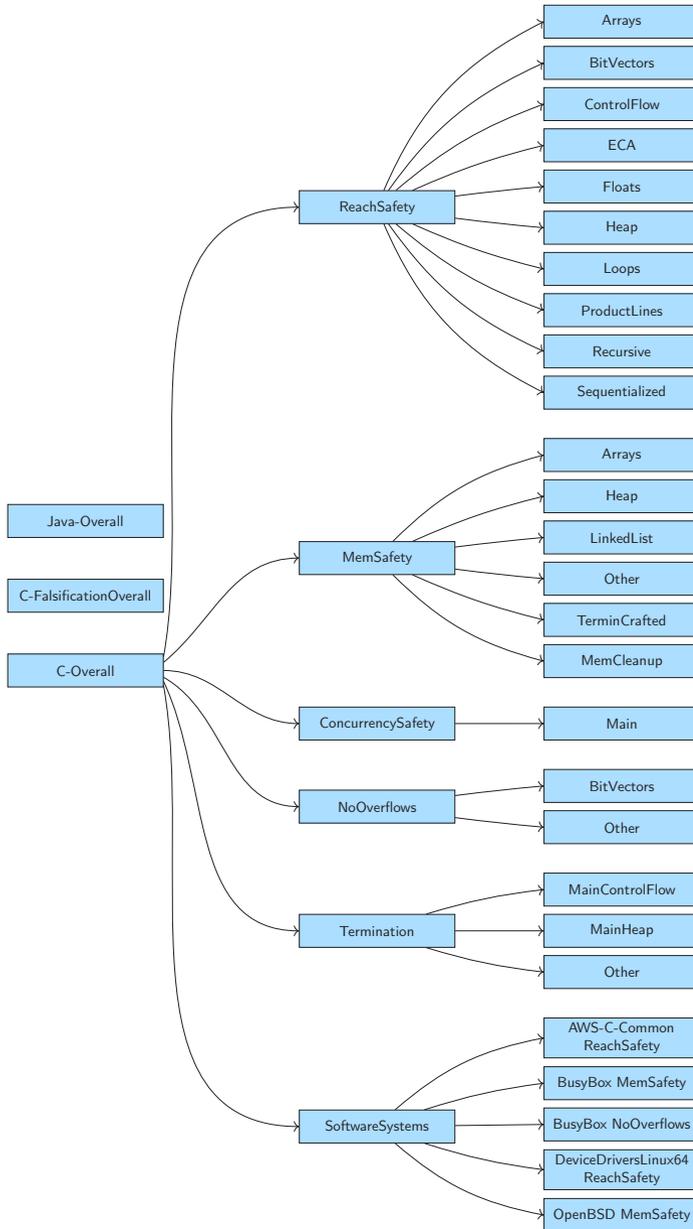


Fig. 2: Category structure for SV-COMP 2020; category *C-FalsificationOverall* contains all verification tasks of *C-Overall* without *Termination*; *Java-Overall* contains all Java verification tasks

Table 1: Properties used in SV-COMP 2020 (unchanged since 2019 [14])

Formula	Interpretation
G ! call(foo())	A call to function <code>foo</code> is not reachable on any finite execution.
G valid-free	All memory deallocations are valid (counterexample: invalid free). More precisely: There exists no finite execution of the program during which an invalid memory deallocation occurs.
G valid-deref	All pointer dereferences are valid (counterexample: invalid dereference). More precisely: There exists no finite execution of the program during which an invalid pointer dereference occurs.
G 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 during which the program lost track of some previously allocated memory.
G valid-memcleanup	All allocated memory is deallocated before the program terminates. In addition to valid-memtrack: There exists no finite execution of the program during which the program terminates but still points to allocated memory. (Comparison to Valgrind: This property can be violated even if Valgrind reports 'still reachable'.)
F end	All program executions are finite and end on proposition <code>end</code> , which marks all program exits (counterexample: infinite loop). More precisely: There exists no execution of the program on which the program never terminates.

Table 2: Scoring schema for SV-COMP 2020 (unchanged since 2017 [13])

Reported result	Points	Description
UNKNOWN	0	Failure to compute verification result
FALSE correct	+1	Violation of property in program was correctly found and a validator confirmed the result based on a witness
FALSE incorrect	-16	Violation reported but property holds (false alarm)
TRUE correct	+2	Program correctly reported to satisfy property and a validator confirmed the result based on a witness
TRUE correct unconfirmed	+1	Program correctly reported to satisfy property, but the witness was not confirmed by a validator
TRUE incorrect	-32	Incorrect program reported as correct (wrong proof)

repository. Table 1 lists the properties and their syntactical representation as overview. Property **G** `valid-memcleanup`, and thus, the category *MemCleanup*, was used for the first time in SV-COMP 2019. The categories *AWS-C-Common* and *OpenBSD* were added for SV-COMP 2020.

The scoring schema is identical for SV-COMP 2017–2020: Table 2 provides the overview and Fig. 3 visually illustrates the score assignment for one property. The scoring schema still contains the special rule for unconfirmed correct results for expected result `TRUE` that was introduced in the transitioning phase: one point is assigned if the answer matches the expected result but the witness was not confirmed.

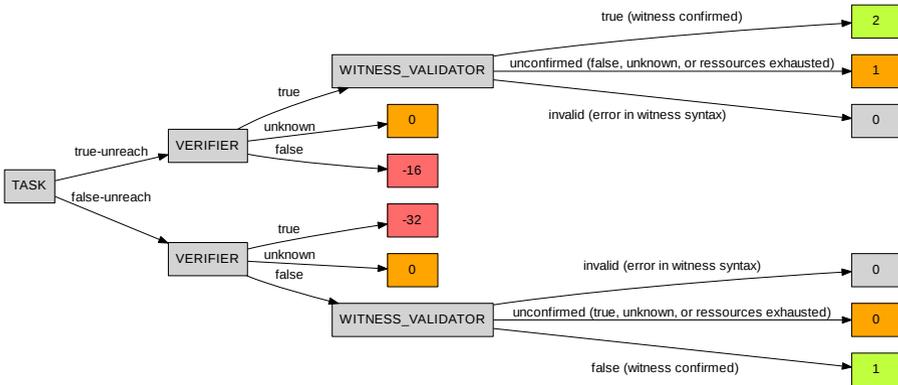


Fig. 3: Visualization of the scoring schema for the reachability property (from [13], © Springer-Verlag)

The ranking was again decided based on the sum of points (normalized for meta categories). In case of a tie, the ranking was decided based on success run time, which is the total CPU time over all verification tasks for which the verifier reported a correct verification result. *Opt-out from Categories* and *Score Normalization for Meta Categories* was done as described previously [9] (page 597).

3 Reproducibility

All major components used in the competition are available in public version repositories. This allows independent replication of the SV-COMP experiments. An overview of the components that contribute to the reproducible setup of SV-COMP is provided in Fig. 4, and the details are given in Table 3. The SV-COMP 2016 report [12] describes all components of the SV-COMP organization and how we ensure that all parts are publicly available for maximal replicability.

We have published the competition artifacts at Zenodo to guarantee their long-term availability and immutability. These artifacts comprise the verification tasks, the produced competition results, and the produced verification witnesses. The DOIs and references are given in Table 4. The archive for the competition results includes the raw results in BENCHEXEC’s XML exchange format, the log output of the verifiers and validators, and a mapping from files names to SHA-256 hashes. The hashes of the files are useful for validating the exact contents of a file, and accessing the files inside the archive that contains the verification witnesses.

To provide a more transparent way of accessing the exact versions of the verifiers that were used in the competition, all verifier archives are stored in a public Git repository. GITLAB was used to host the repository for the verifier archives due to its generous repository size limit of 10 GB. The final size of the Git repository is 5.78 GB.

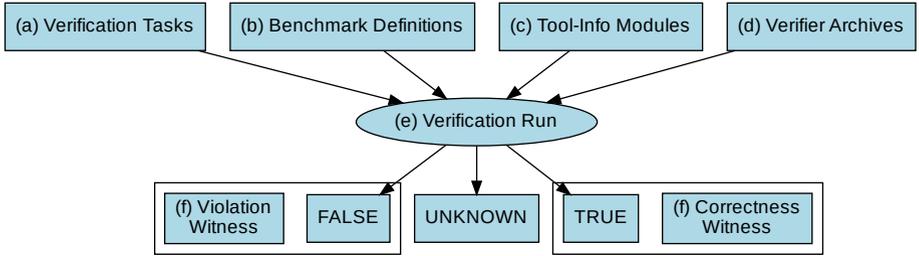


Fig. 4: SV-COMP components and the execution flow

Table 3: Publicly available components for replicating SV-COMP 2020

Component	Fig. 4	Repository	Version
Verification Tasks	(a)	github.com/sosy-lab/sv-benchmarks	svcomp20
Benchmark Definitions	(b)	github.com/sosy-lab/sv-comp	svcomp20
Tool-Info Modules	(c)	github.com/sosy-lab/benchexec	2.5.1
Verifier Archives	(d)	gitlab.com/sosy-lab/sv-comp/archives-2020	svcomp20
Benchmarking	(e)	github.com/sosy-lab/benchexec	2.5.1
Witness Format	(f)	github.com/sosy-lab/sv-witnesses	svcomp20

4 Results and Discussion

The results of the competition experiments represent the state of the art in fully automatic software-verification tools. The report shows the results, in terms of effectiveness (number of verification tasks that can be solved and correctness of the results, as accumulated in the score) and efficiency (resource consumption in terms of CPU time). The results are presented in the same way as in last years, such that the improvements compared to last year are easy to identify. The results presented in this report were inspected and approved by the participating teams. We now discuss the highlights of the results.

Participating Verifiers. Table 5 and the competition web site¹⁴ provide an overview of the participating verification systems. Table 6 lists the algorithms and techniques that are used in the verification tools.

Computing Resources. The resource limits were the same as in the previous competitions [12]: Each verification run was limited to 8 processing units (cores), 15 GB of memory, and 15 min of CPU time. The witness validation was limited to 2 processing units, 7 GB of memory, and 1.5 min of CPU time for violation witnesses and 15 min of CPU time for correctness witnesses. The machines for running the experiments are part of a compute cluster that consists of

¹⁴ <https://sv-comp.sosy-lab.org/2020/systems.php>

Table 4: Artifacts published for SV-COMP 2020

Content	DOI	Reference
Verification Tasks	10.5281/zenodo.3633334	[17]
Competition Results	10.5281/zenodo.3630205	[16]
Verification Witnesses	10.5281/zenodo.3630188	[18]

Table 5: Competition candidates with tool references and representing jury members

Participant	Ref.	Jury member	Affiliation
2LS	[26, 55]	Viktor Malík	BUT, Brno, Czechia
BRICK		Lei Bu	Nanjing U., China
CBMC	[51]	Michael Tautschnig	Amazon Web Services, UK
COASTAL	[67]	Willem Visser	Stellenbosch U., South Africa
CPA-BAM-BNB	[3, 68]	Vadim Mutilin	ISP RAS, Russia
CPA-LOCKATOR	[4]	Pavel Andrianov	ISP RAS, Russia
CPA-SEQ	[22, 36]	Martin Spiessl	LMU Munich, Germany
DARTAGNAN	[40, 53]	Hernán Ponce de León	Bundeswehr U. Munich, Germany
DIVINE	[6, 52]	Henrich Lauko	Masaryk U., Czechia
ESBMC	[38, 39]	Felipe R. Monteiro	Federal U. of Amazonas, Brazil
GACAL	[61]	Benjamin Quiring	Northeastern U., USA
JAVA-RANGER	[65]	Vaibhav Sharma	U. of Minnesota, USA
JAYHORN	[49, 50]	Philipp Ruemmer	Uppsala U., Sweden
JBMC	[33, 34]	Peter Schrammel	U. of Sussex, UK
JDART	[54, 56]	Falk Howar	TU Dortmund, Germany
LAZY-CSEQ	[47, 48]	Omar Inverso	Gran Sasso Science Inst., Italy
MAP2CHECK	[63, 64]	Herbert Rocha	Federal U. of Roraima, Brazil
PeSCo	[35, 62]	Cedric Richter	Paderborn U., Germany
PINAKA	[30]	Saurabh Joshi	IIT Hyderabad, India
PREDATORHP	[44, 59]	Veronika Šoková	BUT, Brno, Czechia
SPF	[57, 60]	Willem Visser	Amazon, USA
SYMBIOTIC	[28, 29]	Marek Chalupa	Masaryk U., Czechia
UAUTOMIZER	[42, 43]	Matthias Heizmann	U. of Freiburg, Germany
UKOJAK	[27, 58]	Alexander Nutz	U. of Freiburg, Germany
UTAIPAN	[37, 41]	Daniel Dietsch	U. of Freiburg, Germany
VERIABS	[1, 2]	Priyanka Darke	Tata Consultancy Services, India
VERIFUZZ	[31]	Raveendra Kumar M.	Tata Consultancy Services, India
YOGAR-CBMC	[70, 71]	Liangze Yin	Nat. U. of Defense Techn., China

168 machines; each verification run was executed on an otherwise completely unloaded, dedicated machine, in order to achieve precise measurements. Each machine had one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 18.04 with Linux kernel 4.15). We used `BENCHEXEC` [23] to measure and control computing resources (CPU time, memory, CPU energy) and `VERIFIERCLOUD`¹⁵ to distribute, install, run, and clean-up verification runs,

¹⁵ <https://vcloud.sosy-lab.org>

Table 6: Algorithms and techniques that the competition candidates offer

Participant	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms
2LS				✓	✓			✓	✓		✓							✓
BRICK	✓		✓	✓	✓			✓	✓									
CBMC				✓							✓							
COASTAL			✓															
CPA-BAM-BnB	✓	✓					✓				✓	✓	✓	✓				
CPA-LOCKATOR	✓	✓					✓				✓	✓	✓	✓				
CPA-SEQ	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓				✓
DARTAGNAN				✓			✓				✓							
DIVINE			✓				✓				✓							
ESBMC				✓	✓						✓							
GACAL																		
JAVA-RANGER			✓								✓							
JAYHORN	✓	✓				✓		✓					✓	✓				
JBMC				✓							✓							
JDART			✓								✓							
LAZY-CSEQ				✓							✓							
MAP2CHECK				✓							✓							
PESCO	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓				✓
PINAKA			✓	✓							✓							
PREDATORHP									✓									
SPF			✓						✓									✓
SYMBIOTIC			✓					✓	✓		✓							
UAUTOMIZER	✓	✓									✓		✓	✓	✓	✓	✓	
UKOJAK	✓	✓									✓		✓	✓	✓	✓	✓	
UTAIPAN	✓	✓					✓	✓			✓		✓	✓	✓	✓	✓	
VERIABS	✓			✓	✓		✓	✓										✓
VERIFUZZ				✓				✓										
YOGAR-CBMC	✓			✓							✓		✓					✓

and to collect the results. The values for time and energy are accumulated over all cores of the CPU. To measure the CPU energy, we use CPU ENERGY METER [24] (integrated in BENCHEXEC [23]).

One complete verification execution of the competition consisted of 138 074 verification runs (each verifier on each verification task of the selected categories according to the opt-outs), consuming 491 days of CPU time and 130 kWh of CPU energy (without validation). Witness-based result validation required 684 858 validation runs (each validator on each verification task for categories with witness validation, and for each verifier), consuming 311 days of CPU time. Each tool was executed several times, in order to make sure no installation issues occur during the execution. Including preruns, the infrastructure managed a total of 1 018 781 verification runs consuming 4.8 years of CPU time, and 10 705 227 validation runs consuming 6.9 years of CPU time.

Quantitative Results. Table 7 presents the quantitative overview of all tools and all categories. The head row mentions the category, the maximal score for the category, and the number of verification tasks. The tools are listed in alphabetical order; every table row lists the scores of one verifier. We indicate the top three candidates by formatting their scores in bold face and in larger font size. An empty table cell means that the verifier opted-out from the respective main category (perhaps participating in subcategories only, restricting the evaluation to a specific topic). More information (including interactive tables, quantile plots for every category, and also the raw data in XML format) is available on the competition web site¹⁶ and in the results artifact (see Table 4).

Table 8 reports the top three verifiers for each category. The run time (column ‘CPU Time’) and energy (column ‘CPU Energy’) refer to successfully solved verification tasks (column ‘Solved Tasks’). We also report the number of tasks for which no witness validator was able to confirm the result (column ‘Unconf. Tasks’). The columns ‘False Alarms’ and ‘Wrong Proofs’ report the number of verification tasks for which the verifier reported wrong results, i.e., reporting a counterexample when the property holds (incorrect FALSE) and claiming that the program fulfills the property although it actually contains a bug (incorrect TRUE), respectively.

Score-Based Quantile Functions for Quality Assessment. We use score-based quantile functions [9, 23] because these visualizations make it easier to understand the results of the comparative evaluation. The web site¹⁶ and the results archive (see Table 4) include such a plot for each category. As an example, we show the plot for category *C-Overall* (all verification tasks) in Fig. 5. A total of 11 verifiers participated in category *C-Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [9]). A more detailed discussion of score-based quantile plots, including examples of what insights one can obtain from the plots, is provided in previous competition reports [9, 12].

¹⁶ <https://sv-comp.sosy-lab.org/2020/results>

Table 8: Overview of the top-three verifiers for each category (measurement values for CPU time and energy rounded to two significant digits)

Rank	Verifier	Score	CPU Time (in h)	CPU Energy (in kWh)	Solved Tasks	Unconf. Tasks	False Alarms	Wrong Proofs
<i>ReachSafety</i>								
1	VERIABS	5543	150	1.6	3 412	171		
2	CPA-SEQ	4396	72	.75	2 700	54	8	
3	PESCo	4376	39	.38	2 518	36	4	
<i>MemSafety</i>								
1	PREDATORHP	611	.78	.010	392	15		
2	SYMBIOTIC	516	.51	.010	358	6		
3	CPA-SEQ	355	.76	.010	264	1		
<i>ConcurrencySafety</i>								
1	LAZY-CSEQ	1279	6.7	.090	1 023	44		
2	YOGAR-CBMC	1275	.39	.000	1 024	33		
3	CPA-SEQ	996	12	.11	830	102		
<i>NoOverflows</i>								
1	CPA-SEQ	483	.93	.010	321	8		
2	UAUTOMIZER	466	1.4	.010	326	0		
3	UTAIPAN	461	1.5	.010	323	0		
<i>Termination</i>								
1	UAUTOMIZER	2942	15	.16	1 606	7		
2	CPA-SEQ	1720	16	.17	1 247	7		
3	2LS	1264	3.2	.030	955	361	3	
<i>SoftwareSystems</i>								
1	SYMBIOTIC	954	.25	.000	676	36	3	1
2	CPA-SEQ	746	21	.24	1 381	363	1	
3	CPA-BAM-BNB	602	8.0	.070	1 411	582	3	4
<i>FalsificationOverall</i>								
1	CPA-SEQ	2772	45	.45	2 240	139	9	
2	SYMBIOTIC	1828	27	.35	1 461	10	3	
3	ESBMC	1639	14	.18	1 819	385	16	
<i>Overall</i>								
1	CPA-SEQ	9219	120	1.3	6 743	535	9	
2	UAUTOMIZER	8178	83	.84	5 523	693	71	2
3	PESCo	8023	120	1.2	6 402	242	32	
<i>JavaOverall</i>								
1	JAVA-RANGER	549	1.3	.010	376			
2	JBMC	527	.18	.000	376			
3	JDART	524	.26	.000	374			

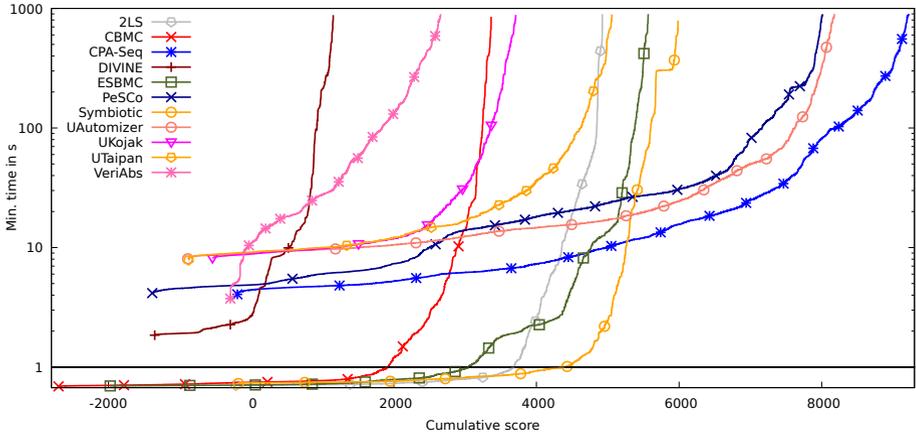


Fig. 5: Quantile functions for category *C-Overall*. Each quantile function illustrates the quantile (x -coordinate) of the scores obtained by correct verification runs below a certain run time (y -coordinate). More details were given previously [9]. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s.

Alternative Rankings. The community suggested to report a couple of alternative rankings that honor different aspects of the verification process as complement to the official SV-COMP ranking. Table 9 is similar to Table 8, but contains the alternative ranking categories *Correct* and *Green Verifiers*. Column ‘Quality’ gives the score in score points, column ‘CPU Time’ the CPU usage of successful runs in hours, column ‘CPU Energy’ the CPU usage of successful runs in kWh, column ‘Solved Tasks’ the number of correct results, column ‘Wrong Results’ the sum of false alarms and wrong proofs in number of errors, and column ‘Rank Measure’ gives the measure to determine the alternative rank.

Correct Verifiers — Low Failure Rate. The right-most columns of Table 8 report that the verifiers achieve a high degree of correctness (all top three verifiers in the C track have less than 2% wrong results). The winners of category *Java-Overall* produced not a single wrong answer. The first category in Table 9 uses a failure rate as rank measure: $\frac{\text{number of incorrect results}}{\text{total score}}$, the number of errors per score point (E/sp). We use E as unit for number of incorrect results and sp as unit for total score. It is remarkable to see that the worst result was 0.38 E/sp in SV-COMP 2019 and is now improved to 0.032 E/sp , with is an order of magnitude better.

Green Verifiers — Low Energy Consumption. Since a large part of the cost of verification is given by the energy consumption, it might be important to also consider the energy efficiency. The second category in Table 9 uses the energy consumption per score point as rank measure: $\frac{\text{total CPU energy}}{\text{total score}}$, with the unit J/sp . It is interesting to see that the worst result from SV-COMP 2019 was 4200 J/sp , and now it is improved to 2200 J/sp .

Table 9: Alternative rankings; quality is given in score points (sp), CPU time in hours (h), energy in kilojoule (kJ), wrong results in errors (E), rank measures in errors per score point (E/sp), joule per score point (J/sp), and score points (sp)

Rank	Verifier	Quality (sp)	CPU Time (h)	CPU Energy (kWh)	Solved Tasks	Wrong Results (E)	Rank Measure (E/sp)
<i>Correct Verifiers</i>							(E/sp)
1	CPA-SEQ	9219	120	1.3	6743	9	.0010
2	UKOJAK	3710	48	0.49	2405	4	.0011
3	2LS	4924	27	0.24	3044	8	.0016
worst							.032
<i>Green Verifiers</i>							(J/sp)
1	CBMC	3365	15	0.16	3217	67	170
2	2LS	4924	27	0.24	3044	8	180
3	ESBMC	5567	35	0.41	5520	51	270
worst							2200

Table 10: Confirmation rate of verification witnesses in SV-COMP 2020

Result	TRUE				FALSE			
	Total	Confirmed	Unconf.	Total	Confirmed	Unconf.		
2LS	2060	2049	99%	11	1449	995	69%	454
CBMC	1949	1821	93%	128	2095	1396	67%	699
CPA-SEQ	4347	3958	91%	389	2931	2785	95%	146
DIVINE	811	793	98%	18	1099	672	61%	427
ESBMC	3779	3701	98%	78	2204	1819	83%	385
PeSCo	3777	3704	98%	73	2867	2698	94%	169
SYMBIOTIC	2196	2146	98%	50	1996	1879	94%	117
UAUTOMIZER	4135	4029	97%	106	2081	1494	72%	587
UKOJAK	1811	1801	99%	10	606	604	100%	2
UTAIPAN	2496	2438	98%	58	1308	730	56%	578
VERIABS	3908	3387	87%	521	1536	1332	87%	204

Verifiable Witnesses. All SV-COMP verifiers are required to justify the result (TRUE or FALSE) by producing a verification witness (except for those categories for which no witness validator is available). We used six independently developed witness-based result validators [19, 20, 21, 25, 66].

The majority of witnesses that the verifiers produced can be confirmed by the results-validation process. Interestingly, the confirmation rate for the TRUE results is significantly higher than for the FALSE results. Table 10 shows the confirmed versus unconfirmed results: the first column lists the verifiers

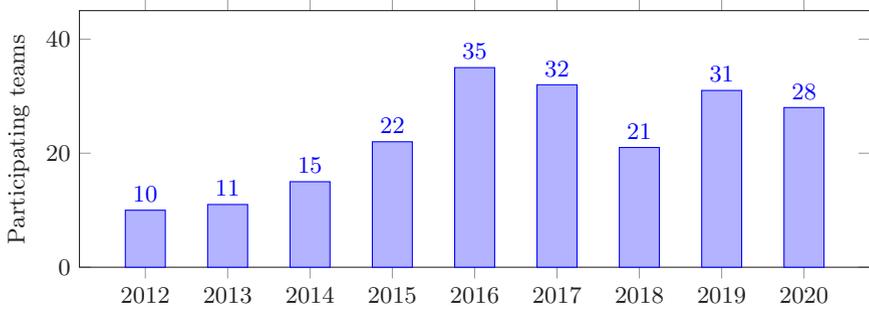


Fig. 6: Number of participating teams for each year

of category *C-Overall*, the three columns for result TRUE reports the total, confirmed, and unconfirmed number of verification tasks for which the verifier answered with TRUE, respectively, and the three columns for result FALSE reports the total, confirmed, and unconfirmed number of verification tasks for which the verifier answered with FALSE, respectively. More information (for all verifiers) is given in the detailed tables on the competition web site¹⁶ and in the results artifact; all verification witnesses are also contained in the witnesses artifact (see Table 4). Result validation is an important topic also in other competitions (e.g., in the SAT competition [5, 69]).

5 Conclusion

SV-COMP 2020, the 9th edition of the Competition on Software Verification, attracted 28 participating teams from 11 countries (see Fig. 6 for the participation numbers). SV-COMP continues to offer a broad overview of the state of the art in automatic software verification. The competition does not only execute the verifiers and collect results, but also validates the verification results, using six independently developed results validators. The number of verification tasks was increased to 11 052 in C and to 416 in Java. As before, the large jury and the organizer made sure that the competition follows the high quality standards of the TACAS conference, in particular with respect to the important principles of fairness, community support, and transparency.

Data Availability Statement. The verification tasks and results of the competition are published at Zenodo, as described in Table 4. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Fig. 4 and Table 3. Furthermore, the results are presented online on the competition web site for easy access: <https://sv-comp.sosy-lab.org/2020/results/>.

References

1. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VERIABS: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141 (2019). <https://doi.org/10.1109/ASE.2019.00121>
2. Afzal, M., Chakraborty, S., Chauhan, A., Chimdyalwar, B., Darke, P., Gupta, A., Kumar, S., M., C.B., Unadkat, D., Venkatesh, R.: VERIABS: Verification by abstraction and test generation (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
3. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BNB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS. pp. 355–359. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_22
4. Andrianov, P., Mutilin, V., Khoroshilov, A.: Predicate abstraction based configurable method for data race detection in Linux kernel. In: Proc. TMPA. CCIS 779, Springer (2018). https://doi.org/10.1007/978-3-319-71734-0_2
5. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT Competition 2016: Recent developments. In: Proc. AAI. pp. 5061–5063. AAI Press (2017)
6. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkal, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Proc. ATVA. pp. 201–207. LNCS 10482, Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
7. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS (3). pp. 3–24. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_1
8. Beyer, D.: Competition on software verification (SV-COMP). In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_38
9. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
10. Beyer, D.: Status report on software verification (Competition summary SV-COMP 2014). In: Proc. TACAS. pp. 373–388. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_25
11. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
12. Beyer, D.: Reliable and reproducible competition results with BENCHEXEC and witnesses (Report on SV-COMP 2016). In: Proc. TACAS. pp. 887–904. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
13. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS. pp. 331–349. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20
14. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
15. Beyer, D.: First international competition on software testing (Test-Comp 2019). Int. J. Softw. Tools Technol. Transf. (2020)

16. Beyer, D.: Results of the 9th International Competition on Software Verification (SV-COMP 2020). Zenodo (2020). <https://doi.org/10.5281/zenodo.3630205>
17. Beyer, D.: SV-Benchmarks: Benchmark set of 9th Intl. Competition on Software Verification (SV-COMP 2020). Zenodo (2020). <https://doi.org/10.5281/zenodo.3633334>
18. Beyer, D.: Verification witnesses from SV-COMP 2020 verification tools. Zenodo (2020). <https://doi.org/10.5281/zenodo.3630188>
19. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). <https://doi.org/10.1145/2950290.2950351>
20. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). <https://doi.org/10.1145/2786805.2786867>
21. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
22. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
23. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). <https://doi.org/10.1007/s10009-017-0469-y>
24. Beyer, D., Wendler, P.: CPU ENERGY METER: A tool for energy-aware algorithms engineering. In: Proc. TACAS (2). LNCS 12079, Springer (2020)
25. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: unpublished manuscript (2020)
26. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9
27. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. Fundam. Inform. **89**(4), 369–392 (2008)
28. Chalupa, M., Jašek, T., Tomovič, L., Hruška, M., Šoková, V., Ayaziová, P., Strejček, J., Vojnar, T.: SYMBIOTIC 7: Integration of PREDATOR and more (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
29. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
30. Chaudhary, E., Joshi, S.: PINAKA: Symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS (3). pp. 234–238. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_20
31. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VERIFUZZ: Program-aware fuzzing (competition contribution). In: Proc. TACAS (3). pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
32. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. JSAT **9**, 207–242 (2016)
33. Cordeiro, L.C., Kesseli, P., Kröning, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: Proc. CAV. pp. 183–190. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_10
34. Cordeiro, L.C., Kröning, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode (competition contribution). In: Proc. TACAS (3). pp. 219–223. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_17

35. Czech, M., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Predicting rankings of software verification tools. In: Proc. SWAN. pp. 23–26. ACM (2017). <https://doi.org/10.1145/3121257.3121262>
36. Dangl, M., Löwe, S., Wendler, P.: CPACHECKER with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
37. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: ULTIMATE TAIPAN with symbolic interpretation and fluid abstractions (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
38. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k -induction and invariant inference (competition contribution). In: Proc. TACAS (3). pp. 209–213. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15
39. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k -induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (Feb 2017). <https://doi.org/10.1007/s10009-015-0407-9>
40. Gavrilenko, N., Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Proc. CAV. pp. 355–365. LNCS 11561, Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19
41. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS. pp. 128–147. LNCS 10422, Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
42. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: ULTIMATE AUTOMIZER and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2). pp. 447–451. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
43. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
44. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: PREDATOR shape analysis tool suite. In: Hardware and Software: Verification and Testing. pp. 202–209. LNCS 10028, Springer (2016). <https://doi.org/10.1007/978-3-319-49052-6>
45. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In: Proc. ISO LA. pp. 608–614. LNCS 7609, Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_45
46. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012: A program verification competition. STTT **17**(6), 647–657 (2015). <https://doi.org/10.1007/s10009-015-0396-8>
47. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: A lazy sequentialization tool for C (competition contribution). In: Proc. TACAS. pp. 398–401. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
48. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proc. PPOPP. ACM (2020)
49. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JAYHORN: A framework for verifying Java programs. In: Proc. CAV. pp. 352–358. LNCS 9779, Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_19

50. Kahsai, T., Rümmer, P., Schäfer, M.: JAYHORN: A Java model checker (competition contribution). In: Proc. TACAS (3). pp. 214–218. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_16
51. Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
52. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_17
53. de Leon, H.P., Furbach, F., Heljanko, K., Meyer, R.: DARTAGNAN: Bounded model checking for weak memory models (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
54. Luckow, K.S., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., Raman, V.: JDART: A dynamic symbolic analysis framework. In: Proc. TACAS. pp. 442–459. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_26
55. Malík, V., Schrammel, P., Vojnar, T.: 2LS: Heap analysis and memory safety (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
56. Mues, M., Howar, F.: JDART: Dynamic symbolic execution for Java bytecode (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
57. Noller, Y., Păsăreanu, C.S., Le, X.B.D., Visser, W., Fromherz, A.: Symbolic PATHFINDER for SV-COMP (competition contribution). In: Proc. TACAS (3). pp. 239–243. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_21
58. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks (competition contribution). In: Proc. TACAS. pp. 458–460. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44
59. Peringer, P., Šoková, V., Vojnar, T.: PREDATORHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
60. Păsăreanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic PATHFINDER: integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Software Eng.* **20**(3), 391–425 (2013). <https://doi.org/10.1007/s10515-013-0122-2>
61. Quiring, B., Manolios, P.: GACAL: Conjecture-based verification (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
62. Richter, C., Wehrheim, H.: PESCO: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
63. Rocha, H.O., Menezes, R., Cordeiro, L., Barreto, R.: MAP2CHECK: Using symbolic execution and fuzzing (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
64. Rocha, H., Barreto, R.S., Cordeiro, L.C.: Memory management test-case generation of C programs using bounded model checking. In: Proc. SEFM. pp. 251–267. LNCS 9276, Springer (2015). https://doi.org/10.1007/978-3-319-22969-0_18
65. Sharma, V., Hussein, S., Whalen, M., McCamant, S., Visser, W.: JAVA RANGER at SV-COMP 2020 (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
66. Svejda, J., Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NITWIT. In: Proc. TACAS. LNCS , Springer (2020)

67. Visser, W., Geldenhuys, J.: COASTAL: Combining concolic and fuzzing for Java (competition contribution). In: Proc. TACAS (2). LNCS 12079, Springer (2020)
68. Volkov, A.R., Mandrykin, M.U.: Predicate abstractions memory modeling method with separation into disjoint regions. Proceedings of the Institute for System Programming (ISPRAS) **29**, 203–216 (2017). [https://doi.org/10.15514/ISPRAS-2017-29\(4\)-13](https://doi.org/10.15514/ISPRAS-2017-29(4)-13)
69. Wetzler, N., Heule, M.J.H., Jr., W.A.H.: DRAT-TRIM: Efficient checking and trimming using expressive clausal proofs. In: Proc. SAT. pp. 422–429. LNCS 8561, Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31
70. Yin, L., Dong, W., Liu, W., Li, Y., Wang, J.: YOGAR-CBMC: CBMC with scheduling constraint based abstraction refinement (competition contribution). In: Proc. TACAS. pp. 422–426. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_25
71. Yin, L., Dong, W., Liu, W., Wang, J.: On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. Softw. Eng. (2018). <https://doi.org/10.1109/TSE.2018.2864122>

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

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





2LS: Heap Analysis and Memory Safety (Competition Contribution)*

Viktor Malík^{**3} , Peter Schrammel^{1,2} , and
Tomáš Vojnar³ 

¹Diffblue Ltd, Oxford, UK

²University of Sussex, Brighton, UK

³FIT, Brno University of Technology, Brno, CZ



Abstract 2LS is a framework for analysis of sequential C programs based on the CPROVER infrastructure and template-based synthesis techniques for checking both safety and termination. The paper presents the main improvements done in 2LS since 2018, which concern mainly the way 2LS handles dynamically allocated objects and structures as well as combinations of abstract domains.

1 Overview

2LS is a static analysis and verification tool for sequential C programs. At its core, it uses the *kIkI* algorithm (*k*-invariants and *k*-induction) [1], which integrates bounded model checking, *k*-induction, and abstract interpretation into a single, scalable framework. *kIkI* relies on incremental SAT solving in order to find proofs and refutations of assertions, as well as to perform termination analysis [2].

The 2019 and 2020 competition versions of 2LS feature product and power abstract domain combinations supporting invariant inference for programs manipulating shape and content of dynamic data structures [4]. Moreover, the 2020 version came with further enhancements for handling advanced features of memory allocation and made a step towards a support of generic abstract domain combinations.

Architecture. The architecture of 2LS has been described in previous competition contributions [7,5]. In brief, 2LS is built upon the CPROVER infrastructure [3] and thus uses *GOTO programs* as the internal program representation. The analysed program is translated into an acyclic, over-approximate single static assignment (SSA) form, in which loops are cut at the edges returning to the loop head. Subsequently, 2LS refines this over-approximation by computing inductive invariants in various abstract domains represented by parametrised logical formulae, so-called templates [1]. The competition version uses the zones domain for numerical variables combined with our shape domain for pointer-typed variables. The SSA form is bit-blasted into a propositional formula and given to a SAT solver. The *kIkI* algorithm then incrementally amends the formula to perform loop unwindings and invariant inference based on template-based synthesis [1].

* The Czech authors were supported by the project 20-07487S of the Czech Science Foundation.

** Jury member: imalik@fit.vut.cz.

2 New Features

The major improvements of 2LS since 2018 are mostly related to analysis of heap-manipulating programs. We build on the shape domain presented in 2018 [5] and introduce abstract domain combinations that allow us to analyse both shape and content of dynamic data structures. Furthermore, we introduce a special handling for the case when an address of a freed heap object is re-used for the next allocation.

Apart from an improved verification of heap-manipulating programs, we also introduce a generic skeleton of an abstract domain join algorithm, which is a step towards a support of generic abstract domain combinations.

2.1 Combinations of Abstract Domains

The capability of 2LS to jointly analyse shape and content of dynamic data structures takes advantage of the template-based synthesis engine of 2LS. Invariants are computed in various abstract domains where each domain has the form of a template while relying on the analysis engine to handle the domain combinators.

Memory model In our memory model, we represent dynamically allocated objects by so-called *abstract dynamic objects*. Each such object is an abstraction of a number of concrete dynamic objects allocated by the same `malloc` call [4].

Shape Domain For analysing the shape of the heap, we use an improved version of the shape domain that we introduced in 2018 [5]. The domain over-approximates the *points-to* relation between pointers and symbolic addresses of memory objects in the analysed program: for each pointer-typed variable and each pointer-typed field of an abstract dynamic object p , we compute the set of all addresses that p may point to [4].

Template Polyhedra Domain For analysing numerical values, we use the template polyhedra abstract domains, particularly the *interval* and the *zones* domains [1].

Shape and Polyhedra Domain Combination Since both domains have the form of a template formula, we simply use them side-by-side in a product domain combination—the resulting formula is a conjunction of the two template formulae [4].

This combination allows 2LS to infer, e.g., invariants describing an unbounded singly-linked list whose nodes contain values between 1 and 10. We show an example of such a list in Figure 1. Here, all list nodes are abstracted by a single abstract dynamic object ao_1 (i.e. we assume that they are all allocated at the same program location). The invariant inferred by 2LS for such a list might look as follows:

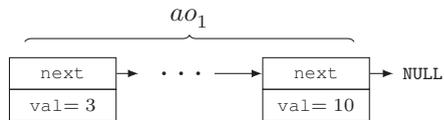


Figure 1. Unbounded singly-linked list abstracted by an abstract dynamic object ao_1 .

$$(ao_1.next = \&ao_1 \vee ao_1.next = \text{NULL}) \wedge ao_1.val \in [1, 10].$$

The first disjunction describes the shape of the list—the *next* field of each node points to some node of the list or to `NULL`¹. The second part of the conjunct is then an invariant in the interval domain over all values stored in the list—it expresses the fact that the value of each node lies in the interval between 1 and 10.

2.2 Symbolic Paths

To improve precision of the analysis, we let 2LS compute different invariants for different *symbolic paths* taken by the analysed program. We require a symbolic path to express which loops were executed at least once. This allows us to distinguish situations when an abstract dynamic object does not represent any really allocated object and hence the invariant for such abstract dynamic object is not valid [4].

The symbolic path domain allows us to iteratively compute a set of symbolic paths p_1, \dots, p_n (represented by guard variables in the SSA) with associated shape and data invariants I_1, \dots, I_n . The aggregated invariant is then $p_1 \Rightarrow I_1 \wedge \dots \wedge p_n \Rightarrow I_n$, which corresponds to a power domain combination.

2.3 Re-using Freed Memory Object for Next Allocations

In C, it is possible that, after a `free` is called, the freed memory is subsequently re-used when a `malloc` is called afterwards. Due to this, it may happen that the error state in the program in Figure 2 is reachable. This situation is, however, difficult to handle for 2LS as its memory model creates a unique abstract dynamic object for each `malloc` call. To overcome this limitation, we have introduced a special variable *fr* that is non-deterministically set to the value of the freed pointer at each `free` call. If two pointers x, y are compared in the analysed program using a relational operator *op*, we transform the comparison $x \text{ op } y$ into

```
int *a = malloc(sizeof(int));
free(a);
int *b = malloc(sizeof(int));
if (a == b)
    // error state
```

Figure 2. Re-using a freed object

$$(x \text{ op } y) \leftrightarrow ((x \neq fr \vee nondet_x) \wedge (y \neq fr \vee nondet_y)). \quad (1)$$

Here, $nondet_x$ and $nondet_y$ are unconstrained boolean variables modelling a non-deterministic choice. If neither x nor y has been freed, then the result of Eq. (1) is equal to $x \text{ op } y$, but if either of the pointers might have been freed, then the result of Eq. (1) is non-deterministic, which makes our analysis sound for the described case.

2.4 Generic Abstract Domain Templates

As is mentioned in Section 1, abstract domains are represented in 2LS by so-called templates. The main reason of templates is that they reduce the second-order problem of finding an inductive invariant to a first-order problem of finding values of template parameters. Apart from defining the form of the template (a parametrised logical formula), each abstract domain also needs to specify an algorithm to perform join of the current

¹ Here, $ao_1.f$ is an abstraction of the f fields of all concrete objects represented by ao_1 . Analogously, $\&ao_1$ is an abstraction of symbolic addresses of all represented objects.

values of template parameters with a model of satisfiability returned by an SMT solver. However, most of the domains use a similar approach to this algorithm, and therefore adding a new abstract domain to 2LS requires one to write an algorithm whose skeleton has already been written in existing domains.

To overcome this problem, we proposed a generic algorithm suitable for all existing abstract domains (see [6] for details). The main idea is based on the fact that most of the templates are conjunctions of multiple formulae, where each has its own parameter and describes a part of the analysed program, e.g., properties of a single program variable.

While this extension did not bring any additional functionality that would increase the score of 2LS in this year's edition of SV-COMP, it opened up possibilities for future enhancements, in particular (1) it simplifies adding of new abstract domains capable of analysing program properties that 2LS is currently not able to handle and (2) it is a significant step towards a support of generic abstract domain combinations that would allow 2LS to arbitrarily combine abstract domains and therefore analyse complex properties of programs requiring simultaneous reasoning in multiple domains.

3 Strengths and Weaknesses

One of the main strengths of 2LS is verification of programs requiring joint reasoning about shape and content of dynamic data structures. In 2019, we contributed 10 benchmarks into the ReachSafety category requiring such reasoning. The domain combination described in Section 2.1 allows 2LS to successfully verify 9 out of 10 of these benchmarks (the last one has timed out), making it the only tool capable of this apart from the category winner. Also, 2LS is notably strong in analysing termination, which is supported by the third place in the Termination category.

Still, there remain a lot of challenges and limitations. The main problem is that 2LS still lacks reasoning about array contents, and that it does not yet support recursion.

4 Tool Setup

The competition submission is based on 2LS version 0.8.² The archive contains the binaries needed to run 2LS (`2ls-binary`, `goto-cc`), and so no further installation is needed. There is also a wrapper script `2ls` which is used by Benchexec to run the tools over the verification benchmarks. See the wrapper script also for the relevant command line options given to 2LS. The further information about the contents of the archive could be found in the `README` file. The tool info module for 2LS is called `two_ls.py` and the benchmark definition file `2ls.xml`. As a back end, the competition submission of 2LS uses Glucose 4.0. 2LS competes in all categories except Concurrency and Java.

5 Software Project

2LS is implemented in C++ and it is maintained by Peter Schrammel with contributions by the community.³ It is publicly available at <http://www.github.com/diffblue/2ls> under a BSD-style license.

² Executable available at <https://doi.org/10.5281/zenodo.3678347>.

³ <https://github.com/diffblue/2ls/graphs/contributors>

References

1. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety Verification and Refutation by k -Invariants and k -Induction. In: Proc. of SAS'15. LNCS, vol. 9291. Springer (2015)
2. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Bit-Precise Procedure-Modular Termination Proofs. TOPLAS 40 (2017)
3. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Proc. of TACAS'04. LNCS, vol. 2988. Springer (2004)
4. Malík, V., Hruška, M., Schrammel, P., Vojnar, T.: Template-Based Verification of Heap-Manipulating Programs. In: Proc. of FMCAD'18. IEEE (2018)
5. Malík, V., Martiček, Š., Schrammel, P., Srivas, M., Vojnar, T., Wahlang, J.: 2LS: Memory Safety and Non-termination (Competition Contrib.). In: Proc. of TACAS'18. Springer (2018)
6. Marušák, M.: Generic Template-Based Synthesis of Program Abstractions. Master's thesis, Brno University of Technology (2019), <https://www.fit.vut.cz/study/thesis/21674/>
7. Schrammel, P., Kroening, D.: 2LS for Program Analysis (Competition Contribution). In: Proc. of TACAS'16. LNCS, vol. 9636. Springer (2016)

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

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





COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution)

Willem Visser^{id}* and Jaco Geldenhuys^{id}

Stellenbosch University, Stellenbosch, South Africa
{visserw,geld}@sun.ac.za



Abstract. COASTAL is a program analysis tool for Java programs. It combines concolic execution and fuzz testing in a framework with built-in concurrency, allowing the two approaches to cooperate naturally.

1 Verification Approach and Software Architecture

COASTAL analyses Java bytecode with an approach that combines concolic execution and fuzz testing in a unified framework. It uses the ASM bytecode manipulation library [2] to add code to compiled class files to monitor and interact with the system under test (SUT). The concurrent COASTAL components that carry out the analysis are shown in Figure 1:

- Multiple *divers* (for concolic analysis) execute the SUT with different concrete input values. A diver run is triggered when a vector of concrete input values is added to the diver input queue d_in . As a diver executes, the instrumented code mirrors the state of the program with symbolic values. At the end of the run, the symbolic path condition that corresponds to the execution is enqueued in the diver output queue d_out .
- Multiple *surfers* (for fuzzing analysis) also execute the SUT with concrete input values. A surfer run is triggered when a vector of concrete input values is added to the surfer input queue s_in . As a surfer executes, lighter instrumentation records the “shape” of the execution path, and at the end of the run, this information is enqueued in the surfer output queue s_out .
- One or more strategies remove and process the information that appears on the diver and surfer output queues. For example, a strategy may remove a path condition, negate one or more constraints, invoke an SMT solver to find input values that will explore the modified path, and enqueue them on d_in or s_in or both. Instrumentation injects the input values into the SUT.
- To share information between components, discovered execution paths are stored in a shared execution tree known as the *pathtree*. The pathtree keeps track of which sub-trees have been fully explored. The pathtree data structure allows for efficient concurrent updates.

* Jury member

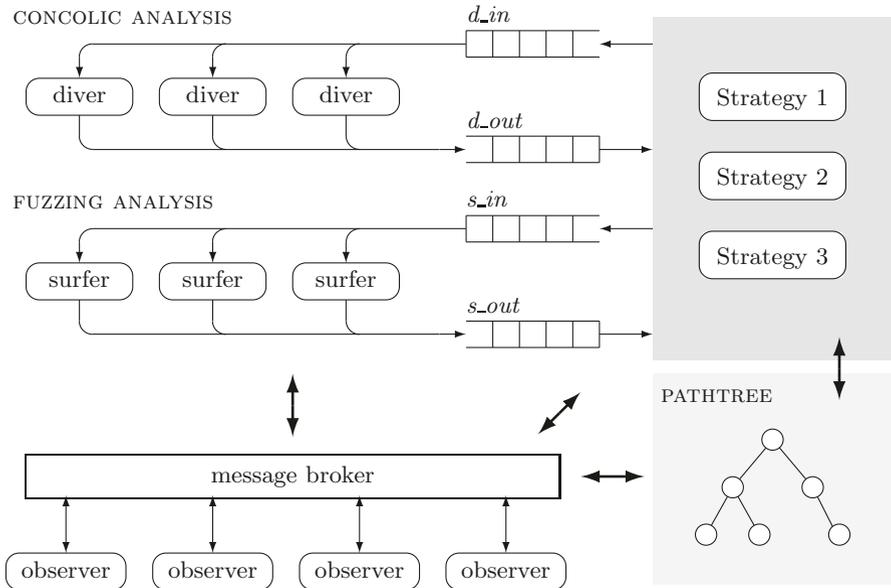


Fig. 1. COASTAL architecture

- Divers, surfers, strategies, and the pathtree signal their actions via a publish-subscribe system. When events are published to the message broker, one or more observers are notified. The observers may, in turn, emit messages that direct the operation of COASTAL.

1.1 Strategies

As an example, a *depth-first strategy* is a simple configuration of COASTAL where the strategy employs only a single diver. The diver produces one path condition that is processed by the strategy by negating the last (deepest) constraint, and sending it to an SMT solver, which produces new input values (if any) that will explore the modified path. If a modified path condition is unsatisfiable, the last constraint is discarded and the process repeats. All path conditions are added to the pathtree as they are discovered. At the end of the analysis, the pathtree contains a summary of the execution tree of the SUT.

Other strategies include *breadth-first* and *random* exploration. Like depth-first exploration, these strategies use only one diver and explore one path condition at a time. On the other hand, a *generational strategy* negates all the constraints of a path condition, one by one, and produces many potential input values. In this case, multiple divers can be used concurrently. Users can also deploy multiple strategies at the same time.

Fuzzing strategies. The user can employ surfers to perform straightforward fuzz testing (in the style of AFL [1,5,6]). Surfers use very little instrumentation.

Unlike the divers — that instrument every bytecode instruction — only the outcomes of branching points are recorded. The “path condition” produced by a surfer is therefore a series of (mostly binary) choices that can be added to the pathtree; it lacks any details about the reason for the choice (for example, instead of “ $x > 5$ ” it may simply record “*false*”), but the shape of the path is preserved. Multiple divers and multiple surfers are deployed concurrently and operate interactively.

Hybrid strategies. More advanced strategies can combine concolic and fuzzing analysis to exploit the strengths of both approaches: surfers (fuzzing) can rapidly explore new territory of the execution space, while divers (concolic) can investigate hard-to-reach corners. Such hybrid strategies enqueue (semi-)random inputs on `s_in` and the results contribute to a “skeletal” pathtree. Since surfers produce results at a high rate, the easy-to-explore parts of the execution space are more quickly saturated. Unexplored regions of the pathtree are passed to the divers, and their results, in turn, open up new regions for the surfers to explore.

1.2 Observers and Models

COASTAL was designed with extensibility in mind. One example is the use of *observers*. Any component is allowed to subscribe to the various message streams, and can interact with the system by publishing messages of their own, or by making direct calls to the public COASTAL API. Examples of observer tasks include:

- monitor assertions and halt COASTAL when they are violated,
- record instruction, line, and condition coverage,
- enforce assumptions and prune undesired execution paths,
- gather information and display progress in a GUI.

In theory, strategies themselves could be implemented as observers. But since they are central to the operation of COASTAL, they are given special treatment.

Users can replace system- or user-level libraries by more appropriate *models*, either as a whole or on a method-by-method basis. For example, a complex library implementation of `String.substring()` can be replaced with a simpler, more efficient model that produces the same result and the same symbolic constraints.

2 Strengths and weaknesses

The tool’s strength lies in the combination of concolic and fuzzing analysis, but COASTAL is still under development and a “deep” bug (now fixed) prevented the use of fuzzing. Participation in SV-COMP [3] was invaluable in this regard: Several bugs and missing functionality were revealed and corrected.

Results. COASTAL does not output any incorrect answers, but produces an *unknown* result in 19% of cases. This is shown in column “Count” below.

Answer	Count		Immediate	
<i>true</i>	135	32.45%	121	89.63%
<i>false</i>	202	48.57%	134	66.34%
<i>unknown</i>	79	18.99%	27	34.18%
All	416	100%	282	67.79%

For many cases, the answer is produced instantaneously (column “Immediate”). In the case of *unknown* answers, this indicates that COASTAL aborted its analysis because of an as-yet unsupported feature such as symbolic array sizes. For the $79 - 27 = 52$ non-immediate *unknown* answers, COASTAL timed out because of large search spaces.

The longest-running *true* answer required 2 diver runs, each taking 20.48sec (`printtokens_eqchk.yml`), whereas the longest-running *false* answer required 141 diver runs, each taking 0.54sec (`spec1-5_product1.yml`). This highlights a fundamental weakness of the tool: a long-running SUT takes longer to analyse. A generational strategy where multiple divers execute concurrently can ameliorate this problem, but on average does not find errors as quickly as the breadth-first strategy. This points to the need to refine the generational strategy to prioritize shallow unexplored paths.

3 Tool setup

Download. <http://doi.org/10.5281/zenodo.3679243> [7]

Configuration. COASTAL is configured to use a breath-first search strategy and a single diver. Z3 [4] is set as the constraint solver. (It is the only external tool required to run COASTAL and a Linux executable version is included in the download above.) Path conditions are limited to 800 conjuncts, and a time limit of 240 second is set. Symbolic strings are limited to 25 characters. Custom models are used for some Java classes: `Character`, `String`, `StringBuilder`, `Pattern`, `Matcher`, `Scanner`. COASTAL competed in the `JavaOverall` category.

Installation. The download above is self-contained. The COASTAL project at <https://github.com/DeepseaPlatform/coastal/> includes shell scripts to package and run COASTAL for SV-COMP in the `extra/svcomp` subdirectory. The scripts needs an external copy of the Z3 solver to be available.

4 Software Project

COASTAL is developed by the authors at Stellenbosch University, South Africa. It is available at <https://github.com/DeepseaPlatform/coastal/> and is distributed under the GNU Lesser General Public License version 3.

References

1. American Fuzzy Lop, <http://lcamtuf.coredump.cx/afl/>. Accessed 11 Jan 2020
2. ASM Library, <https://asm.ow2.io/>. Accessed 10 Jan 2020
3. Beyer, D.: Advances in Automatic Software Verification: SV-COMP 2020. In: Biere, A., Parker, D. (eds.) TACAS 2020 (2), LNCS, vol. 12079. Springer, Heidelberg (2020).
4. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008, LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008).
5. Miller, B.P., Fredriksen, L., So, B.: An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM* **33**(12), 32–44, (1990)
6. Sutton, M., Greene, A., Amini, P.: *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional (2007).
7. Visser, W., Geldenhuys, J.: (2020, February 22). COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution). <http://doi.org/10.5281/zenodo.3679243> Zenodo. (2020)

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

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





DARTAGNAN: Bounded Model Checking for Weak Memory Models (Competition Contribution)

Hernán Ponce-de-León^{*1} , Florian Furbach²,
Keijo Heljanko³, and Roland Meyer²



¹University of the Bundeswehr Munich, Munich, Germany

²TU Braunschweig, Braunschweig, Germany

³University of Helsinki and HIIT, Helsinki, Finland

Abstract. DARTAGNAN is a bounded model checker for concurrent programs under weak memory models. What makes it different from other tools is that the memory model is not hard-coded inside DARTAGNAN but taken as part of the input. For SV-COMP'20, we take as input sequential consistency (i.e. the standard interleaving memory model) extended by support for atomic blocks. Our point is to demonstrate that a universal tool can be competitive and perform well in SV-COMP. Being a bounded model checker, DARTAGNAN's focus is on disproving safety properties by finding counterexample executions. For programs with bounded loops, DARTAGNAN performs an iterative unwinding that results in a complete analysis. The SV-COMP'20 version of DARTAGNAN works on BOOGIE code. The C programs of the competition are translated internally to BOOGIE using SMACK.

1 Overview and Software Architecture

DARTAGNAN is a bounded model checker for concurrent programs under weak memory models. It expects as input a program P annotated with a reachability condition S , a memory model \mathcal{M} , and an unrolling bound k . It recursively unwinds all loops in P up to the bound k . The unwound program is converted into an SMT formula that symbolically represents all candidate executions. The memory model will filter out some candidates using a second formula, we explain this below. Events of a candidate execution model (instances of) program instructions, like memory accesses, local computations, and conditional/unconditional jumps. Edges model relations between events, including *program order* (the order within a thread), *data-dependencies* (an assigned variable is used within an expression), *reads-from* (matching each read with the write from which it takes its value), and *coherence* (the order in which writes commit to the memory).

A memory model can be understood as a predicate over candidate executions that declares some of them valid. We describe memory models in the CAT language [2]. A memory model is defined as a set of relations (those mentioned

* Jury member.



Fig. 1. CAT model used for SV-COMP’20.

above and others derived as unions, transitive/reflexive closures, compositions, etc.) and constraints over them (emptiness, acyclicity and irreflexivity). Given a memory model, we construct a formula that evaluates to true precisely under the candidate executions that are valid according to the memory model. [Figure 1](#) shows the memory model used for SV-COMP’20. To support atomic blocks, DARTAGNAN adds a specific edge (*rrw*) for every pair of events between `VERIFIER_atomic_begin()` and its matching `VERIFIER_atomic_end()` or in a `VERIFIER_atomic_` function. We encode atomicity for sequential consistency (SC) as the empty intersection of *rrw* and paths starting and ending with an external communication (i.e. between different threads). This means once an atomic block starts, external communications with the block are forbidden until all events in the block have been executed.

DARTAGNAN comes with a rich assertion language inspired by HERD [1]. Assertions define inequalities over the values of local and global variables. They can be used freely throughout the code, rather than being limited to the end of the execution. Semantically, our assertions do not stop the execution but record the failure and continue. To achieve this, each instructions `assert(exp)` is transformed to a local computation $\mathbf{f} \leftarrow \text{exp}$ where the fresh variable $\mathbf{f} \in \mathbf{F}$ stores the value of `exp` at the corresponding point of the execution. We refer to the formula $\bigvee_{\mathbf{f} \in \mathbf{F}} \neg \mathbf{f}$ as the reachability condition.

The formula for candidate executions of the program, the formula for validity under the given memory model, and the reachability condition together (in conjunction) yield the SMT encoding of the reachability problem at hand. Any solution to the conjunction corresponds to an execution that is valid according to the memory model and violates at least one assertion. Details on the encoding can be found in [8,9].

DARTAGNAN implements a may-alias analysis to improve pointer precision and a novel relation analysis. The latter technique reduces the SMT encoding to those parts of the relations that might affect the consistency with the memory model, resulting in a considerably smaller formula. Relation analysis improves the performance up to two orders of magnitude [4,5]. We remark that related approaches represent each candidate execution explicitly [1,6]. Thanks to the symbolic representation of executions and static analysis techniques such as relation analysis, DARTAGNAN is often more efficient [4,5].

[Figure 2](#) shows the overall architecture of DARTAGNAN. It reads programs written in the litmus format of HERD [1] or the intermediate verification language BOOGIE [7]. For the competition, C programs are compiled to LLVM and then

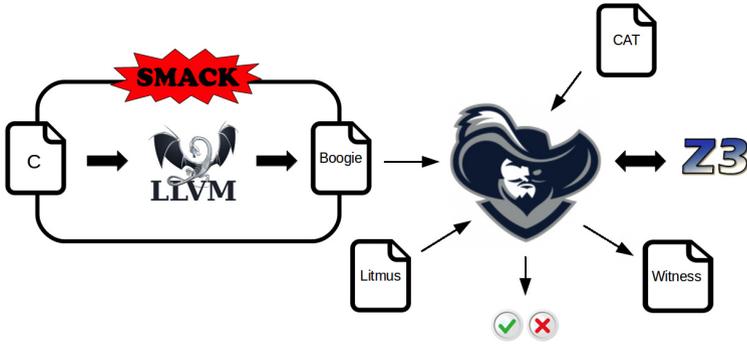


Fig. 2. DARTAGNAN’s architecture.

translated internally to BOOGIE using the SMACK tool [11]. The SMT solver is Z3 [3]. When a violation is found, DARTAGNAN returns a witness execution.

2 Strengths and Weaknesses

The main strength of DARTAGNAN is its fully configurable memory model. Unfortunately, in SV-COMP’20 there is no category for verification tasks under weak memory models. On the SV-COMP’20 benchmarks, DARTAGNAN reports only one incorrect result, being beaten in that aspect only by CPACHECKER, DIVINE, LAZY-CSEQ and YOGAR-CBMC; three of them category winners. The incorrect result is related to the use of pointer arithmetic which is currently not supported by our alias analysis.

Its main strength is also its main weakness: DARTAGNAN’s performance cannot quite match that of other verifiers that were developed specifically for sequential consistency. DARTAGNAN performs particularly poor on benchmarks with big atomics blocks. This is the case for most of the verification tasks in the `pthread-wmm` group which represent 83% of the ConcurrencySafety category. The problem is that DARTAGNAN adds `rmw` edges for all pairs in an atomic block. This results in a large encoding (even using relation analysis) and highly impacts its performance.

3 Tool Setup and Configuration

Besides the program to be verified, DARTAGNAN expects a CAT file containing the memory model of interest. For SV-COMP’20, this is the extension of sequential consistency given in Figure 1. The tool is run by executing the following command:

```
$ java -jar dartagnan/target/dartagnan-V-jar-with-dependencies.jar
  -cat <CAT file> -i <program file> [options]
```

Placeholder `V` is the tool version (currently 2.0.5) and `options` is used to configure the unrolling bound, the alias analysis, and the fixpoint encoding. The full list of options can be found on the project website (see [Section 4](#)).

To make sure not to miss a violation, the competition version of DARTAGNAN implements an iterative approach. Initially, the bounded model checking algorithm is called with an unrolling bound of one. If it finds a violation or can prove that all loops have been unrolled completely (this is done using unwinding assertions), the verification process terminates with a conclusive answer. If not, DARTAGNAN increases the bound by one and repeats the process. For program with an infinite state space, our tool does not terminate.

DARTAGNAN participates in the ConcurrencySafety category. No specification file is required. The artifact is available on [\[10\]](#). To reproduce the results of the competition, the tool can be executed with the following wrapper script:

```
$ Dartagnan-SVCOMP.sh <program file>
```

4 Software Project and Contributors

The project home page is <https://github.com/hernanponcedeleon/Dat3M>. DARTAGNAN is open source software distributed under the MIT license.

Acknowledgement: We thank Dirk Beyer and Philipp Wendler for their help during the process of integrating DARTAGNAN into the competition framework. We also thank Natalia Gavrilenko for her contributions to the development of the bounded model checking engine of the tool [\[4,5\]](#).

References

1. The herdttools7 tool suite. <https://github.com/herd/herdttools7>.
2. Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language CAT. *CoRR*, abs/1608.07531, 2016.
3. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
4. Natalia Gavrilenko. Improving scalability of bounded model checking for weak memory models. Master’s thesis, Aalto University, Department of Computer Science, 2019.
5. Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In *CAV*, volume 11561 of *LNCS*, pages 355–365. Springer, 2019.
6. Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. Cerberus-BMC: A principled reference semantics and exploration tool for concurrent and sequential C. In *CAV*, volume 11561 of *LNCS*, pages 387–397. Springer, 2019.
7. K. Rustan M. Leino. This is Boogie 2. 2008.
8. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In *SAS*, volume 10422 of *LNCS*, pages 299–320. Springer, 2017.

9. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC with memory models as modules. In *FMCAD*, pages 1–9. IEEE, 2018.
10. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Replication package for the Dartagnan tool for SVCOMP 2020. <http://dx.doi.org/10.5281/zenodo.3678318>, February 2020.
11. Zvonimir Rakamaric and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, volume 8559 of *LNCS*, pages 106–113. Springer, 2014.

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

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





VeriAbs : Verification by Abstraction and Test Generation (Competition Contribution)

Mohammad Afzal¹, Supratik Chakraborty² , Avriti Chauhan¹, Bharti Chimdyalwar¹,
Priyanka Darke^{1,*}, Ashutosh Gupta², Shrawan Kumar¹, Charles Babu M³,
Divyesh Unadkat^{1,2} , and R Venkatesh¹

¹ Tata Research Development and Design Center, Pune, India

² Indian Institute of Technology, Bombay, India

³ Chennai Mathematical Institute, Chennai, India

Abstract. VeriAbs is a strategy selection based reachability verifier for C code. It analyzes the structure of loops, and intervals of inputs to choose one of the four verification strategies implemented in VeriAbs. In this paper, we present VeriAbs version 1.4 with updates in three strategies. We add an array verification technique called *full-program induction*, and enhance the existing techniques of loop pruning, *k*-path interval analysis, and disjunctive loop summarization. These changes have improved the verification of programs with arrays, and unstructured loops and unstructured control flows.

1 Verification Approach

VeriAbs is a reachability checker for C code that employs a portfolio of techniques and works by smartly selecting a sequence of techniques for each problem instance. Specifically, it performs structural and interval analysis of the input code to determine a sequence of suitable verification techniques, or a strategy [2]. An earlier version of the tool appeared in [9]. Figure 1 shows the architecture with this year's enhancements in dashed lines. When the input program contains unstructured loops, VeriAbs performs fuzz testing in parallel with *k*-induction. If the program does not contain unstructured loops but loops manipulating arrays, VeriAbs applies array abstraction techniques like loop shrinking, loop pruning, and full-program induction [7] in sequence. If the program contains inputs of very short ranges, VeriAbs applies explicit state model checking, and loop invariant generation using program behaviour, syntax and counter-examples in parallel [2]. Otherwise VeriAbs applies *k*-path interval analysis, loop abstraction, loop summarization, bounded model checking, and *k*-induction in the order presented in the architecture. If any technique successfully (in)validates the encoded properties, the tool reports the result, generates the witness, and exits. We next explain the enhancements made to VeriAbs this year.

1.1 Tool Enhancements

Full-Program Induction. VeriAbs applies full-program induction as presented in [7] to programs manipulating arrays of a symbolic size N given as a parameter. It takes as input

* Jury member, corresponding author : priyanka.darke@tcs.com

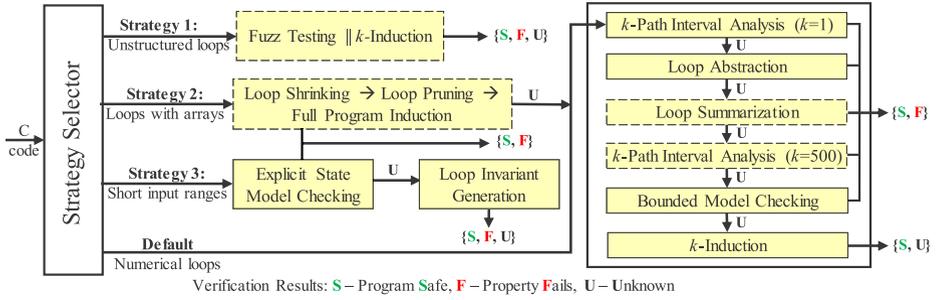


Fig. 1. Architecture Diagram

a parameterized program represented by P_N , annotated with parameterized pre- and post-conditions represented by $\varphi(N)$ and $\psi(N)$ respectively and checks the validity of the Hoare triple $\{\varphi(N)\} P_N \{\psi(N)\}$ for all values of $N (> 0)$. We summarize the technique in [7] here.

In the base case, it verifies that the given Hoare triple holds for a fixed number of values of N (say for $N = 1$). If the check fails, a property violation is reported. It then hypothesizes that the Hoare triple $\{\varphi(N - 1)\} P_{N-1} \{\psi(N - 1)\}$ holds for $N > 1$, where P_{N-1} is the program with parameter $N - 1$. In the induction step, the technique synthesizes a code fragment ∂P_N , called the *difference program*, such that $\{\varphi(N)\} P_N \{\psi(N)\}$ is valid iff $\{\varphi(N)\} P_{N-1}; \partial P_N \{\psi(N)\}$ is valid. The *difference program* is the computation to be performed after the program P_{N-1} has executed to get the same state as P_N . It then computes a formula $\partial\varphi(N)$, called the *difference pre-condition*, such that $\varphi(N)$ is implied by the conjunction of $\varphi(N - 1)$ and $\partial\varphi(N)$, and that $\partial\varphi(N)$ continues to hold after the execution of P_{N-1} . The induction step now needs to prove the validity of $\{\psi(N - 1) \wedge \partial\varphi(N)\} \partial P_N \{\psi(N)\}$. It uses weakest pre-condition computation to infer formulas $pre(N)$ over the variables and arrays whose values were computed by P_{N-1} and subsequently read in ∂P_N . Base case is checked for $pre(N)$ and it is subsequently used to strengthen the pre- and post-conditions in the inductive step. The technique, thus, inducts over the entire program via the parameter N , in place of inducting over individual loops by using specialized predicates as in [6]. Full-program induction does not rely on inductive invariants for each loop in the program.

k-Path Interval Analysis. VeriAbs implements a k -path interval analysis which is an extension of the standard non-relational interval domain [2]. It maintains the path-wise data ranges of variables along a configurable k number of paths at each program point, thus matching the precision of relational domains. When the number of paths at the join point exceeds k , a subset of paths are merged to maintain k paths at the join point. In previous versions, arbitrary subsets of paths were merged. For SV-COMP 2020, the join operation identifies variables of interest (VOIs) with respect to the given property to decide which paths to merge such that VOIs can retain precise values.

```

1 b=0, d=0, c=30;
2 a = *;
3 if (a == 10)
4   c = 30; //Path P1
5 else if (a < 10)
6   b = 3; //Path P2
7 else if (a > 10)
8   d = 31; //Path P3
9 if (c==30 && a==10)
10  d = 31;
11 if (a >= 10)
12  assert(d == 31);
    
```

Fig. 2. Example

Consider the example shown in Figure 2 with a valid property at line 12 to be analyzed with $k=2$ and the VOI d . It can be seen that three paths – P1, P2 and P3 join at line number 9. The enhanced join operation merges paths P1 and P2 so that the resultant paths are as follows:

P1+P2: $\{a=[\text{MIN}, 10], b=[0, 3], c=30, d=0\}$,
 P3: $\{a=[11, \text{MAX}], b=0, c=30, d=31\}$.

This information at the join point helps validate the property. Earlier, the join operation could merge the path P3 with P1 or P2, leading to an imprecise interval $- [0,31]$ of d at the join point, resulting in spurious property violation. Our implementation considers variables used in the encoded property as the VOIs.

Loop Pruning is an array abstraction technique that defines a set of criteria (and a resulting set of program transformation rules) which if satisfied by loops processing arrays, it is sufficient to analyze the first few elements instead of the entire array [14]. In this version, pruning has been extended to programs containing nested loops and multidimensional arrays. By structural analysis, we identify if elements of the multidimensional array are processed *uniformly* in loops. If yes, we compute reduced dimensions of the array (for example, $a[m][m]$ may be reduced to $a[4][4]$). We have also refined the pruning criteria to improve its applicability over multidimensional and dynamically allocated arrays, 56 additional SV-COMP'20 ReachSafety benchmarks are solved by the current implementation of array pruning as compared to the previous version.

Disjunctive Loop Summarization. VeriAbs analyses interleavings of unique paths within a loop to produce its disjunctive summary to find errors and proofs [2]. In the current version, VeriAbs extends this technique in the following situations: (a) while it earlier restricted affine transformations to identity matrices, we now allow diagonal matrices with finite monoid [4]; (b) we use the approach of generating *flattening*s as shown in [4] for loops which are *flattable*; (c) we use VeriAbs' general philosophy of deriving over-approximate summaries using the techniques in [12], when precise disjunctive summary is not derivable.

2 Software Architecture

VeriAbs is primarily developed in Java and Perl. It implements all program analyses (except full-program induction) and program transformers in Prism [13], the TCS Research program analysis framework. It transforms programs processing multidimensional or dynamically allocated arrays in loops to equivalent programs with symbolically sized 1D arrays. This transformed program is consumed by VAJRA v1.0 [7], the tool that implements full-program induction. VAJRA uses LLVM v6.0.0 [15] compiler infrastructure for program transformations and Z3 SMT solver v4.8.7 [10] for checking the validity of Hoare triples and for computing weakest pre-conditions. For BMC VeriAbs uses the C Bounded Model Checker (CBMC) v5.10 [8] with the Glucose Syrup SAT solver v4.0 [3]. For fuzz testing we enhance American Fuzzy Lop [16] to allow test case mutation within valid data ranges generated by k -path interval analysis for better path coverage. VeriAbs uses k -induction with continuously refined invariants as implemented in CPAchecker v1.8 [5] for an improved precision over our existing light weight implementation of k -induction.

In this version, we additionally derive disjunctive invariants for correctness witnesses using abstract acceleration and abstract interpretation, and add them to the control flow automaton generated by CPAchecker. If all implemented techniques fail, we use techniques implemented in Ultimate Automizer v3204b741 [11] to generate correctness witnesses.

3 Strengths and Weaknesses

The main strengths of VeriAbs are (1) strategy selection that correlates strengths of verification techniques and input code properties, and (2) a portfolio of sound techniques. Weaknesses:

(1) long strategies – the lengths of strategies executed by VeriAbs in the worst case can be ten techniques, thus time consuming. Hence, smarter and shorter strategies are needed. (2) Non-linear expressions in loops – loop abstractions in VeriAbs assign non-deterministic values to variables modified in such expressions. (3) Multidimensional arrays in loops manipulating non-contiguous locations – these are limitations of loop shrinking and pruning. These weaknesses are not limitations of the state-of-the-art, and appropriate techniques if integrated into VeriAbs can be easily invoked by the strategy selector to enable verification of such programs.

4 Tool Setup and Configuration

The VeriAbs SV-COMP 2020 executable is available for download at <https://gitlab.com/sosy-lab/sv-comp/archives-2019/tree/master/2020/veriams.zip>. To install the tool, download the archive, extract its contents, and then follow the installation instructions in `VeriAbs/INSTALL.txt`. To execute VeriAbs, the user needs to specify the property file of the respective verification category using the `--property-file` option and the `-64` option for programs with a 64 bit architecture. The witness is generated in the current working directory as `witness.graphml`. A sample command is as follows:

```
VeriAbs/scripts/veriams <-64> --property-file ALL.prp example.c
```

VeriAbs participated in the ReachSafety and the SoftwareSystems-ReachSafety categories of SV-COMP 2020. The BenchExec wrapper script for the tool is `veriams.py` and the benchmark description file is `veriams.xml`.

5 Software Project and Contributors

VeriAbs is maintained by some members of the Foundations of Computing group at TCS Research [1]. They can be contacted at veriams.tool@tcs.com. We are thankful to the developers of American Fuzzy Lop, CBMC, CPAchecker, Glucose Syrup, LLVM, UAutomizer and Z3 for allowing us to use the tools within VeriAbs.

References

1. TCS Research. <http://www.tcs.com/research/Pages/default.aspx>
2. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by Abstraction and Test Generation. In: ASE. pp. 1138–1141 (2019)
3. Audemard, G., Simon, L.: On the glucose sat solver. IJAIT **27**(01) (2018)
4. Bardin, A., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: ATVA. pp. 474–488 (2005)
5. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: CAV. pp. 622–640 (2015)
6. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: SAS. pp. 428–449 (2017)
7. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction. In: TACAS (2020)
8. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS (2004)
9. Darke, P., Prabhu, S., Chimdyalwar, B., Chauhan, A., Kumar, S., Basakchowdhury, A., Venkatesh, R., Datar, A., Medicherla, R.K.: VeriAbs: Verification by Abstraction and Test Generation - (Competition Contribution). In: TACAS. pp. 457–462 (2018)
10. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS. pp. 337–340 (2008)

11. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate automizer and the search for perfect interpolants - (competition contribution). In: TACAS. pp. 447–451 (2018)
12. Jeannot, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. SIGPLAN Not. **49**(1), 529–540 (2014)
13. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: an experience. In: ISEC. pp. 99–102 (2011)
14. Kumar, S.: Scaling up Property Checking. https://www.cse.iitb.ac.in/~as/thesis_soft.pdf (2019)
15. Lattner, C.: LLVM and Clang: Next generation compiler technology. In: The BSD Conference (2008)
16. Zalewski, M.: American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>

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

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





GACAL: Conjecture-based Verification (Competition Contribution)

Benjamin Quiring * and Panagiotis Manolios 

Northeastern University, Boston MA, USA



Abstract. GACAL verifies C programs by searching over the space of possible invariants, using traces of the input program to identify potential invariants. GACAL uses the ACL2s theorem prover to verify these potential invariants, using an interface provided by ACL2s for connecting with external tools. GACAL iteratively searches for and proves invariants of increasing complexity until the program is verified.

1 Verification Approach

GACAL is a tool for verifying reachability queries in C programs by iteratively and efficiently performing conjecture generation and conjecture verification. Conjecture generation involves searching through the space of possible conjectures using evaluation-based testing to identify likely-to-hold conjectures, and conjecture verification consists of using software verification technology to verify these conjectures. Our initial motivation was to develop a computational agent that can automatically complete the Invariant Game [1], in which players suggest invariants that are used by a reasoning engine to verify imperative programs, which we did with success- GACAL is a more fully developed form of the underlying conjecture generation ideas. This section presents a brief overview of GACAL's basic structure and methods for conjecture-based verification, and then discusses these, as well as associated challenges, in more depth. Section 2 provides information about the GACAL project, Section 3 provides an evaluation of GACAL, and Section 4 concludes this paper and discusses future work.

In GACAL, conjectures are potential invariants paired with program locations. Evaluation-based testing consists of evaluating possible invariants using execution-produced program traces. The ACL2s theorem prover [2] verifies conjectures using a graph representation of the input program. To search through the space of conjectures, GACAL first constructs a space of terms, which are C-expressions composed of the constants, variables, and arithmetic/bitwise operators in the program. Terms are combined using relational and logical operators to create possible invariants, and possible invariants which hold in all generated program traces are promoted to potential invariants and turned into conjectures. Discovered potential invariants are then analyzed using ACL2s and, if proven, used to verify the program. In the case that the program cannot be verified from the currently proven invariants, the above process is repeated: construct new, more complex, terms, find potential invariants via testing on program traces,

* Jury member: quiring.b@northeastern.edu

prove potential invariants, attempt program verification, and repeat. At a high-level, this loop is the heart of GACAL's conjecture-based verification.

GACAL's approach to verification presents challenges which can be summarized into two categories: how to minimize the number of generated conjectures, and how to optimize the interactions with ACL2s. The techniques GACAL uses to address these challenges, as well as a more in-depth explanation of the previously mentioned methods are outlined below.

Term and Invariant Construction GACAL builds the space of terms by iteratively constructing all terms of a fixed size, where the size of a term is the number of constants, variables, and operators in that term. GACAL uses a collection of rewrite rules to filter the newly constructed terms: terms which can be rewritten to an equivalent form that has already been constructed are not kept. The size partial order on terms allows GACAL to perform rewriting effectively. Furthermore, the term constructor searches for new rewrite rules by evaluating and comparing terms under a set of random assignments to find pairs of equivalent terms. The discovered equivalences are generalized and turned into rewrite rules which are added to the collection of rewrite rules. We designed the rewriting techniques to have the property that all terms which cannot be rewritten are semantically distinct. In general, the term space is at least asymptotically exponential in size, and the rewriting techniques above, for the class of problems we consider, significantly improve the asymptotics.

Possible invariants are C-expressions of the form $x == y$, $x < y$, $x <= y$, and $P \parallel Q$, where x, y are terms and P, Q are possible invariants. We allow multiple invariants to be associated with each program location, hence, we do not need explicit conjunction. We note that the space of possible invariants is closed under logical negation. GACAL filters out possible invariants which can be rewritten to an equivalent form that has already been created, reducing the size of the invariant space. The order the invariant space is searched over is deterministic and independent of the given program, and was chosen because it worked well for the benchmark programs. At a high level, GACAL inspects more specific invariants before more general invariants (e.g. $x == y$ before $x <= y$).

Trace Generation To produce traces through the program GACAL creates many initial program states which randomly seed the result of all nondeterministic behaviors that occur during execution of the program, making them deterministic. For example, a seeded pseudo-random number generator can obtain values for 'nondeterministic integer' expressions. The initial states are propagated through the program for a bounded number of steps, generating a set of states associated with each program location. These initial traces are not changed during the course of verification.

Testing on program traces is essential to GACAL's conjecture generation, but programs may, for example, contain loops with many iterations or not terminate, and so obtaining traces which correspond to complete program executions may be computationally infeasible or impossible. To address this, GACAL creates

additional types of traces which approximate the input program’s behavior. The first type of these traces generalizes large constants to small and/or nondeterministic values, which allows loops with originally many iterations to be completed. The second type uses the counter-example generation abilities of ACL2s [3,4,5] to generate states at any program location which satisfy all currently proven invariants at that location, which are then propagated through the program. As GACAL proves more invariants, it recomputes the second type of traces to obtain a better approximation of the program. Since invariants tested on these traces are later checked for correctness, the fact that the traces may not reflect the original program’s behavior does not introduce unsoundness. The states from the above two methods are only used to test invariants at a program location if there are no states from the original traces produced for that location, and if traces cannot be found at all then GACAL assumes all invariants are potential.

Conjecture Verification To prove conjectures, GACAL uses an algorithm which takes previously proven invariants as well as currently unproven potential invariants and iteratively removes invariants which cannot be proven until it reaches a fixpoint. This process requires a large number of verification queries and for the majority of programs checking these queries using ACL2s is where the majority of execution time is spent. To improve the ability of ACL2s to reason about GACAL queries, we developed an arithmetic library consisting of ACL2s theorems about the GACAL-supported C operators. Additionally, GACAL caches previous queries and their results, which allows it to answer queries that are similar to cached queries, without using the theorem prover. Finally, GACAL saves counter-examples that ACL2s provides when it falsifies queries and uses them to falsify new queries.

2 Tool Setup and Software Project and Architecture

The competition submission¹ uses GACAL version 1.0. GACAL requires Python 3, Java, and Common Lisp, and the competition archive contains all files necessary to run GACAL without further installation. Other relevant information may be found in the README file. GACAL only competes in the C ReachSafety-Loops category. GACAL is maintained by Benjamin Quiring and Panagiotis Manolios, and is implemented primarily in Common Lisp. The external tools used by GACAL are the Eclipse CDT parser and the ACL2 Sedan [2]. GACAL is publicly available at <https://gitlab.com/acl2s/conjecture-generation/gacal> under a GNU GPLv3 license.

GACAL does not handle all C language features. Most importantly, GACAL does not handle arrays and types other than 32-bit unsigned and signed integers. There is no theoretical reason for this. GACAL does not correctly model C semantics for undefined behavior in signed arithmetic. There is a bug in the contest submission for translating goto statements into our graph representation of programs which affects a small number of benchmarks.

¹ Available at <https://gitlab.com/sosy-lab/sv-comp/archives-2020> and Zenodo [6].

3 Evaluation

GACAL performs best on programs it can execute to completion because this allows us to produce high quality traces covering all program locations. When this is not the case, GACAL often creates false conjectures which lead to a large number of theorem prover queries. Additionally, we note GACAL's execution time depends on the size of the term and invariant spaces, which grow exponentially based on the number of program variables, constants, and operations. The current version of GACAL verifies 66 of the 109 benchmark programs it parses, and the top three tools on this distribution verified 102, 70, and 70. There was one program which no other tools could verify, though GACAL succeeded.

The core of GACAL consists of potential invariant generation using program traces and the rewriting methods as outlined above. We found that the addition of the arithmetic library is essential to our ability to reason about unsigned arithmetic and the mod operator, allowing GACAL to verify 10% more total programs (which deal primarily with the listed features) and cuts the average time to query ACL2s by 33% on the verification queries which were not caught by the caching. We found that the additional trace generation methods did not significantly increase the number of programs that were verified, though they did decrease the average time for verifying a program. The caching of proof results and counter-examples is able to eliminate 85% of all verification queries from being submitted to ACL2s for checking, which increases the number of programs which are verified by over 10% and almost halves the average cost to verify a program. The caching methods also amplifies the benefits of the library and extra trace generation methods.

4 Conclusions and Future Work

There are many ways to improve GACAL, including incorporating classical analyses such as range analysis, abstract interpretation, symbolic evaluation, etc, as well as handling a larger subset of the C language. Another improvement to GACAL is to perform the search for disjunctive invariants more efficiently; currently GACAL often finds many potential but false disjunctive conjectures, which result in a large number of verification queries. One way to improve the search may be to analyze the program to find meaningful hypotheses, which could considerably lower the number of tested and generated conjectures.

We believe that GACAL provides evidence that our conjecture-based verification techniques can be used to improve current software verification tools, as we were able to verify a competitive number of programs on the distribution we parse and we were able to verify a program that all other tools failed to verify, despite not using any of the classical analyses identified above.

References

1. Walter, A., et. al., Gamification of Loop-Invariant Discovery from Code. HCOMP, 2019.
2. Chamarthi, H., Dillinger, P., Manolios, P., Vroon, D. The ACL2 Sedan theorem proving system. TACAS, 2011.
3. Manolios, P. Counterexample Generation Meets Interactive Theorem Proving: Current Results and Future Opportunities. ITP, 2013.
4. Chamarthi, H., et. al. Integrating Testing and Interactive Theorem Proving. ACL2, 2011.
5. Chamarthi, H., Manolios, P. Automated Specification Analysis Using an Interactive Theorem Prover. FMCAD, 2011.
6. Quiring, B., Manolios, P. GACAL v1.0 SV-comp 2020 submission (Version 1.0). Zenodo, 2019. <http://doi.org/10.5281/zenodo.3681607>.

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

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





Java Ranger at SV-COMP 2020 (Competition Contribution)

Vaibhav Sharma^{1*}, Soha Hussein^{1,2}, Michael W. Whalen¹, Stephen McCamant¹, and Willem Visser³

¹ University of Minnesota, Minneapolis, MN, USA
{vaibhav, husse200, mwwhalen, smccaman}@umn.edu

² Ain Shams University, Cairo, Egypt
soha.hussien@cis.asu.edu.eg

³ Stellenbosch University, Stellenbosch, South Africa
visserw@sun.ac.za



Abstract. Path-merging is a known technique for accelerating symbolic execution. One technique, named “veritestng” by Avgerinos et al. uses summaries of bounded control-flow regions and has been shown to accelerate symbolic execution of binary code. But, when applied to symbolic execution of Java code, veritestng needs to be extended to summarize dynamically dispatched methods and exceptional control-flow. Such an extension of veritestng has been implemented in Java Ranger by implementing as an extension of Symbolic PathFinder, a symbolic executor for Java bytecode. In this paper, we briefly describe the architecture of Java Ranger and describe its setup for SV-COMP 2020.

1 Approach

Symbolic execution is a well-known program analysis technique that has been applied to many applications such as test generation [3,7], equivalence checking [6,8], and vulnerability finding [13]. However, when applied to large software, symbolic execution can suffer from scalability challenges caused by path explosion. Path-merging techniques such as veritestng [1] and dynamic state merging [4] help alleviate these scalability limitations. In particular, veritestng attempts to construct a static summary of a multi-path region and use it. Veritestng has been shown to significantly accelerate symbolic execution of binary code. Given that a large amount of software in use today is still written in Java, it is desirable to bring the benefits of veritestng to symbolic execution of Java as well. However, features such as dynamic dispatch make path-merging for Java code challenging [11]. The summary of a multi-path region that contains a dynamically-dispatched method call can only be constructed if the method to be called can also be summarized. Java Ranger (JR) extends the current state-of-the-art path-merging ideas presented by Avgerinos et al. [1] by first building static summaries which are later transformed using runtime information such as

* Jury Member

the dynamic type of an object reference used for accessing a field. Java Ranger is built as an extension to Symbolic PathFinder (SPF) [5].

2 Architecture

Java Ranger is implemented as an SPF listener that watches for symbolic branch conditions in branching instructions. On encountering a symbolic branch instruction, JR attempts to create a summary for the multi-path region that begins at that branch instruction and ends at its exit points. A multi-path region is a region of code that begins at a branch instruction with a symbolic branch condition. An exit point of a multi-path region is either (1) the first program location in a control-flow path through the multi-path region which could not be summarized, or (2) the location of the immediate post-dominator of the multi-path region. This mechanism is also explained by Sharma et al. [12] in Figure 4.

3 Strengths And Weaknesses

Since JR improves scalability limitations of symbolic execution, its strength can only be observed when running it over large software. However, JR falls back to vanilla symbolic execution when it finds no opportunity for path-merging. SV-COMP 2020 had 416 verification tasks in the Java track. More information on SV-COMP 2020 can be found in its competition report [2]. JR instantiated at least one static summary on 96 different benchmarks of the 416 benchmarks. The summary for a multi-path region can be instantiated more than once on each benchmark because it is possible that the symbolic executor will encounter the same multi-path region more than once while running the benchmark. In total, JR instantiated 356 unique summaries. The total number of instantiated summaries used by JR was 20,182. JR also inlined a method summary a total of 62,857 times while instantiating these summaries.

JR also had a “unknown” conclusion on 40 of the 416 SV-COMP 2020 verification tasks. 22 of the 40 were caused due to our JR configuration which turned off support for symbolic strings because we found SPF’s support for solving string constraints was not stable. 9 “unknown” conclusions were reached due to missing support for symbolic array lengths in multi-dimensional arrays. 8 of the 40 occurred due to a timeout. The last “unknown” result occurs in the equivalence check verification task in the ApacheCLI benchmark due to JR’s use of a depth limit.

We made use of two depth limit parameters in SV-COMP 2020. The first was a limit on the exploration depth of our baseline symbolic executor, SPF. The second was a depth limit on the recursive depth to which our method summaries would be inlined. While we wished to avoid the use of any such limit, we found similar kinds of limits were used by many participating tools in SV-COMP 2019. It is common to use some kind of limitation when applying symbolic execution tools in practice, since they can get bogged down by path explosion or related problems, and path-merging helps with but does not eliminate this issue.

The Java verification category of SV-COMP 2020 did not score a tool’s answer differently if it used a depth limit for producing that answer. Instead, the use of depth limit is reflected in each tool’s score only if it caused the tool to produce an incorrect answer. We describe these depth limits and JR’s configuration options in the following section.

4 Tool Setup and Configuration

Java Ranger’s setup is very similar to the setup used by SPF. Since Java Ranger is simply an extension of SPF, the Java Ranger directory can be specified as a valid `jpf-symbc` extension of JPF. A JR configuration requires the following additions.

`veritestestingMode = <1-5>`

`veritestestingMode` specifies the path-merging features to be enabled with each higher number adding a new feature to the set of features enabled by the previous number. Setting `veritestestingMode` to 1 runs vanilla SPF. Setting it to 2 enables path-merging for multi-path regions with no method calls and a single exit point. Setting it to 3 adds path-merging for multi-path regions that make method calls where the method can be summarized by Java Ranger. Setting it to 4 adds path-merging for multi-path regions with more than one exit point caused due to exceptional behavior and unspecialized method calls. Setting it to 5 adds path-merging for summarizing `return` instructions in multi-path regions by treating them as an additional exit point.

`performanceMode = <true or false>`

Setting `performanceMode` to `true` causes Java Ranger to minimize the number of solver calls to check the feasibility of the path condition when summarizing a multi-path region with multiple exit points.

`TARGET_CLASSPATH_WALA=<classpath of target code>`

Java Ranger needs this variable to be set up as environment variable. It is not part of the `.jpf` configuration file. This environment variable tells Java Ranger where it should be expecting to find code that needs to be statically summarized.

`jitAnalysis=<true or false>`

When turned on (the default value), this option causes JR to summarize multi-path regions when it encounters them. When turned off, JR attempts to summarize all multi-path regions reachable in a statically-computed interprocedural call graph up to a configurable limit.

`recursiveDepth=<an integer value>`

This option forces JR to restrict inlining of method summaries up to the value provided for this option. We set this parameter to 12 for SV-COMP 2020.

The following option is a JPF [14] configuration option which we also used for SV-COMP 2020.

`search.depth_limit=<an integer value>`

This option forces JPF to restrict its exploration to the depth provided as the value for this option. JPF constructs a tree of possible choices and explores the tree in a heuristic order, depth-first by default. Since JR is built as an extension

to SPF, which is in turn built as an extension to JPF, we were able to restrict JR's exploration of choices using this option. We set this parameter to the value 13 for SV-COMP 2020.

5 Software Project and Contributors

Java Ranger is an extension of SPF. It is maintained on GitHub [9]. The version of Java Ranger that participated in Sv-COMP 2020 is publicly available [10]. For more information, please contact the authors of this paper.

6 Acknowledgments

The research described in this paper has been supported in part by the National Science Foundation under grant 1563920.

References

1. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing Symbolic Execution with Veritesting. In: Proceedings of the 36th International Conference on Software Engineering. pp. 1083–1094. ICSE 2014, ACM, New York, NY, USA (2014)
2. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). LNCS 12079, Springer (2020), https://www.sosy-lab.org/research/pub/2020-TACAS.Advances_in_Automatic_Software_Verification.SV-COMP_2020.pdf
3. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. ACM, New York, NY, USA (2005)
4. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient State Merging in Symbolic Execution. In: PLDI. pp. 193–204. PLDI '12, ACM, New York, NY, USA (2012)
5. Păsăreanu, C.S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., Rungta, N.: "Symbolic PathFinder: Integrating Symbolic Execution With Model Checking For Java Bytecode Analysis". *Automated Software Engineering* **20**(3), 391–425 (Sep 2013)
6. Ramos, D.A., Engler, D.R.: Practical, Low-effort Equivalence Verification of Real Code. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 669–685. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011)
7. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 263–272. ESEC/FSE-13, ACM, New York, NY, USA (2005)
8. Sharma, V., Hietala, K., McCamant, S.: Finding Substitutable Binary Code By Synthesizing Adaptors. In: 11th IEEE Conference on Software Testing, Validation and Verification (ICST) (Apr 2018)
9. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W.: Java Ranger. <https://github.com/vaibhavbsharma/java-ranger> (2019–2020)
10. Sharma, V., Soha, Michael, Stephen, Willem: Java Ranger at SV-COMP 2020 (Feb 2020). <https://doi.org/10.5281/zenodo.3678718>

11. Sharma, V., Whalen, M.W., McCamant, S., Visser, W.: Veritesting Challenges in Symbolic Execution of Java. In: Java PathFinder Workshop (Jan 2018)
12. Sharma, V., Whalen, M.W., McCamant, S., Visser, W.: Veritesting challenges in symbolic execution of Java. SIGSOFT Softw. Eng. Notes **42**(4), 1–5 (Jan 2018). <https://doi.org/10.1145/3149485.3149491>
13. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: Network and Distributed System Security Symposium (NDSS) (2016)
14. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (Apr 2003). <https://doi.org/10.1023/A:1022920129859>

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

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





JDART: Dynamic Symbolic Execution for JAVA Bytecode (Competition Contribution)

Malte Mues[✉] and Falk Howar[✉]

Dortmund University of Technology
Dortmund, Germany
malte.mues@tu-dortmund.de
falk.howar@tu-dortmund.de



Abstract. JDART performs dynamic symbolic execution of JAVA programs: it executes programs with concrete inputs while recording symbolic constraints on executed program paths. A constraint solver is then used for generating new concrete values from recorded constraints that drive execution along previously unexplored paths. JDART is built on top of the Java PathFinder software model checker and uses the JCONSTRAINTS library for the integration of constraint solvers.

1 Overview

JDART is a dynamic symbolic execution engine for the JVM build on top of Java PathFinder (JPF) [11]. Dynamic symbolic execution [4, 6] (sometimes also referred to as concolic execution) executes programs with concrete values while recording symbolic constraints for execution paths. The approach combines the benefits of fast concrete execution with the possibility of generating new concrete values, triggered by symbolic constraints, that exercise previously unexplored program behaviors. JDART can be used for checking assertions in Java programs: Concolic execution will explore new program paths until either (a) an assertion violation is discovered, (b) all program paths have been explored, or (c) resource limits of the analysis are exhausted.

The initial driver of the development of JDART was the need for an analysis that is robust enough to handle large and complex systems, concretely the AUTORESOLVER software for prediction and resolution of airplane loss of separation developed at NASA Ames Research Center [7]. Though JDART provides a robust and scalable platform for dynamic symbolic analysis of JAVA programs [7], we had to extend its functionality in several ways in order to be able to compete at SV-COMP 2020 [1]. We developed:

1. a new analysis mode in which fresh symbolic variables are introduced during analysis (in contrast to a fixed number of manually declared symbolic values),
2. a number of symbolic models encoding environment behavior (driven by SV-COMP 2020 benchmarks), and
3. a new mode for solving constraints in a sequence of attempts using successively weaker bounds on variables (cf. Section 2).

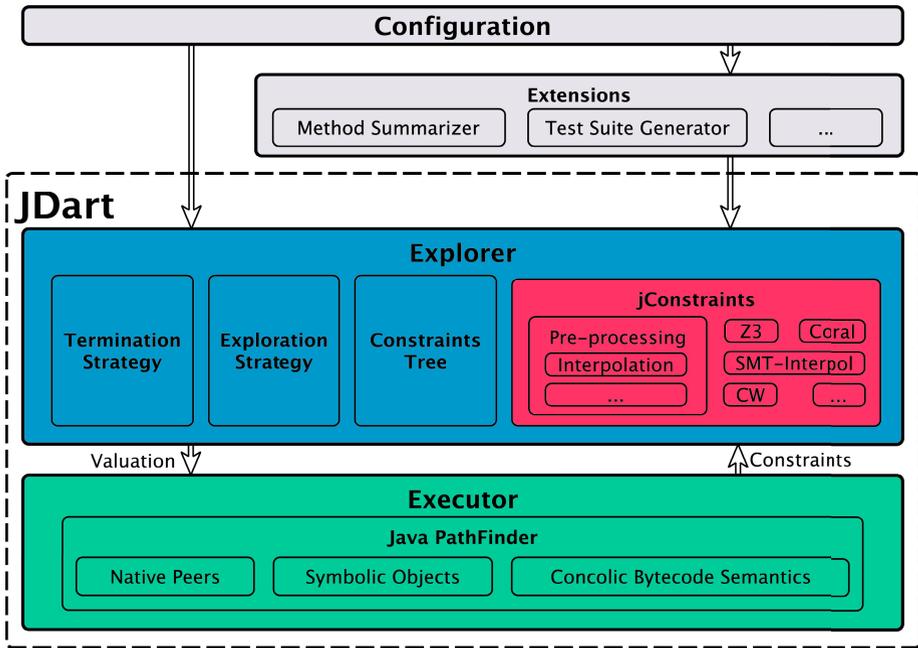


Fig. 1: Architecture of JDART [7].

While (1) enabled JDART to enter the competition, (2) accounts for the largest part of improvements over our own baseline, and (3) contributes to better performance on some benchmarks with assertion violations in big state spaces.

2 Architecture

JDART combines dynamic execution with recording and analysis of symbolic path constraints. It runs as an extension of the JPF software model checker [11]. In particular, JDART uses the JAVA virtual machine implemented by JPF and its capabilities for annotating values on the stack and the heap with symbolic information. The tool itself is written in JAVA and uses JCONSTRAINTS [5] for encoding SMT problems. Moreover, JCONSTRAINTS acts as a frontend to an SMT solver (e.g., Z3 [3]) used for finding concrete values that drive the analysis.

Figure 1 illustrates the architecture of JDART: The tool consists of three layers: Concrete analysis frontends make up the top layer (e.g., generation of method summaries, generation of test suites, assertion checking). The main components record and analyze execution paths (Explorer) and perform concolic execution (Executor). The *Executor* uses concolic implementations of bytecode instructions. These bytecodes are executed instead of the original JPF bytecodes. A concolic bytecode tracks the symbolic representation of a value and annotates a concrete value with its symbolic counterpart. Whenever execution takes a

branching decision based on a concrete value with a symbolic annotation, the symbolic value is added to the constraints tree maintained by the *Explorer*. A constraint solver is used for finding concrete values that drive execution along unexplored paths of the tree.

Leveraging the modular architecture of JDART and JCONSTRAINTS, we implemented a meta-constraint solver for finding small concrete values for symbolic numeric variables. This allows JDART to find assertion violations faster and with less resource consumption in cases where a symbolic variable controls the number or length of execution paths (e.g., symbolic array size or a symbolic loop bound). The meta-constraint solver performs multiple calls to an SMT solver, adding successively weaker bounds to numeric variables. E.g., for a path constraint φ over symbolic numeric variable x , the solver adds bounds $(-z \leq x) \wedge (x \leq z)$ with $z \in (1, 2, 3, 5, 8, 13, 21, \dots)$, i.e., the first numbers in the Fibonacci sequence. If the solver finds a model for the constraint, JDART uses this model for driving concolic execution. In case no model is found in a fixed number of attempts, the SMT solver is called without added bounds. The number of attempts is a configuration parameter of JDART and was fixed to 7 for SV-COMP 2020.

Analysis of JDART can be bounded by termination strategies. When checking assertions the termination strategy is stopping on the first occurrence of an assertion violation. Additional strategies could be bounding depth of the symbolic analysis, bounding runtime, or termination on specific errors. We refer the reader to [7] for a more detailed and complete discussion of the features of JDART.

3 Strengths and Weaknesses

JDART scored 524 points (max. of 602) in the JAVA track and was declared third winner for JAVA, behind JBMC (527 points) [2] and JAVA RANGER (549 points) [9]. All other tools scored considerably fewer points than JDART (next best is COASTAL [10] with 472). As JAVA RANGER and JBMC, JDART did not report a single incorrect verdict. JDART exhibits the general strengths and weaknesses of dynamic and symbolic analysis approaches for JAVA programs:

Runtime. Driven by concrete execution, the analysis is fairly fast. JDART is overall the second fastest tool in cases where it can provide an answer. Not using bounds JDART, on the other hand, has a relatively high number of timeouts and runs that terminate due to resource limitations — and thus only the fourth lowest cumulative runtime.

Symbolic Strings. Particular to JAVA verification is the challenge of providing models for the behavior of classes in the JAVA standard library. In SV-COMP 2020 such models are mostly required for analyzing benchmarks that extensively incorporate String processing. We made a substantial contribution to the code base of JDART and implemented models for `java.lang.String` and related classes. As a consequence, JDART can analyze all but one corresponding benchmark examples (JDART currently cannot analyze regular expressions symbolically).

Unbounded Behavior. Based on principles of symbolic execution, JDART does not terminate on unbounded loops or in case of unbounded recursion, leading to a number of timeouts on the corresponding set of benchmarks.

4 Tool Setup

The source code of JDART used for the competition artifact [8] is available on GitHub¹. JDART is designed as a plug-in to JPF and relies on ant as a build system. One of its dependencies is the `jpf-core` project [11]. The other dependency is the JCONSTRAINTS library, which was configured to use Z3 [3] with incremental solving as a constraint solver for SV-COMP 2020.

For the competition, JDART is wrapped by the `run-jdart.sh` shell script which generates `.jpf` configuration files, specifying which benchmark to analyze and the global configuration options to JDART: For SV-COMP 2020 all termination criteria except for assertion violations are disabled, executing JDART as an almost unbounded assertion checker (the only bound in place is an upper bound of 127 on maximal length of String variables). The shell script records and interprets the output of JDART and can also report the version of JDART.

5 Software Project

The version of JDART that was used in SV-COMP 2020 is maintained by the Automated Quality Assurance Group at Technical University of Dortmund (in particular by the authors of this paper) and is available under the Apache License, version 2.0, on GitHub¹. An initial version of JDART was developed by the authors of [7] at NASA Ames Research Center and Carnegie Mellon University. The original version of JDART is available on GitHub².

Acknowledgments. We are grateful for the work on JDART and JCONSTRAINTS by the respective original authors. Our success would not have been possible without their contributions.

References

1. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). LNCS 12079, Springer (2020), https://www.sosy-lab.org/research/pub/2020-TACAS.Advances_in_Automatic_Software_Verification_SV-COMP_2020.pdf
2. Cordeiro, L., Kroening, D., Schrammel, P.: Jbmc: Bounded model checking for java bytecode. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 219–223. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_17

¹ <https://github.com/tudo-aqua/jdart>,
Commit `c7e30a29b98a69df2c7c96ae39b90ba0fe00e204`

² <https://github.com/psycopaths/jdart>

3. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
4. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM (2005). https://doi.org/10.1007/978-3-642-19237-1_4
5. Howar, F., Jabbour, F., Mues, M.: JConstraints: A library for working with logic expressions in Java. In: Models, Mindsets, Meta: The What, the How, and the Why Not?, pp. 310–325. Springer (2019). https://doi.org/10.1007/978-3-030-22348-9_19
6. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976). <https://doi.org/10.1145/360248.360252>
7. Luckow, K.S., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., Raman, V.: JDart: A dynamic symbolic analysis framework. In: Proceedings of TACAS 2016. pp. 442–459 (2016). https://doi.org/10.1007/978-3-662-49674-9_26
8. Mues, M., Howar, F.: JDart artifact used in SV-COMP 2020. Zenodo (2020). <https://doi.org/10.5281/zenodo.3678593>
9. Sharma, V., Hussein, S., Whalen, M., McCamant, S., Visser, W.: Java Ranger at SV-COMP 2020 (competition contribution). In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12079, pp. 393–397. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_27
10. Visser, W., Geldenhuys, J.: COASTAL: Combining concolic and fuzzing for Java (competition contribution). In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12079, pp. 373–377. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_23
11. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. *Automated Software Engineering* **10**(2), 203–232 (Apr 2003). <https://doi.org/10.1023/A:1022920129859>

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

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





Map2Check: Using Symbolic Execution and Fuzzing

(Competition Contribution)

Herbert Rocha^{1*} , Rafael Menezes³ ,
Lucas C. Cordeiro² , and Raimundo Barreto³



¹Department of Computer Science, Federal University of Roraima, Roraima, Brazil
herbert.rocha@ufrr.br

²Department of Computer Science, University of Manchester, Manchester, United Kingdom

³Institute of Computing, Federal University of Amazonas, Amazonas, Brazil

Abstract. Map2Check is a software verification tool that combines fuzzing, symbolic execution, and inductive invariants. It automatically checks safety properties in C programs by adopting source code instrumentation to monitor data (e.g., memory pointers) from the program’s executions using LLVM compiler infrastructure. For SV-COMP 2020, we extended Map2Check to exploit an iterative deepening approach using LibFuzzer and Klee to check for safety properties. We also use Crab-LLVM to infer program invariants based on reachability analysis. Experimental results show that Map2Check can handle a wide variety of safety properties in several intricate verification tasks from SV-COMP 2020.

1 Overview

Fuzzing involves providing random data as input to a program and then checks for crashes. By contrast, path-based symbolic execution is an entirely static method that symbolically explores the program state-space [1]. Due to a focus on single runs, fuzzing techniques scale up relatively well. Path-based symbolic execution gives more confidence in the verification results, but it suffers from the path-explosion problem, thus limiting scalability. Here we exploit an iterative approach using fuzzing and symbolic execution to implement a tool named Map2Check v7.3.1. Our main original contributions include: (i) use LibFuzzer [7] to provide random data as input to C programs to quickly expose “shallow” bugs, i.e., those that do not require complex data input; (ii) implement a new runtime library and instrumentation approach to monitor for crashes, failing built-in assertions and pointer safety; (iii) adopt Crab-LLVM [11] to infer invariants; (iv) exploit a sequential approach with LibFuzzer and KLEE [3] to check safety properties in a novel way; and (v) adopt MetaSMT as a wrapper around various SMT solvers, e.g., Boolector [2] and Yices [4], previously not supported by our tool. The SV-COMP’20 results show that Map2Check can be useful in both falsifying and proving reachability error and pointer safety-related properties.

* Jury member

2 Verification Approach

Map2Check uses compiler techniques to analyze C programs using LLVM compiler infrastructure, thereby tracking pointer addresses and variable assignments in the LLVM bytecode [8]. In order to hold all values used in the analysis, a container API is employed in Map2Check. The tool also generates *built-in* assertions and checks them adopting an approach with fuzzing (to falsify properties) and symbolic execution (to prove the correctness). Fig. 1 illustrates the Map2Check flow, which has the following main steps: (i) convert the C code into the LLVM IR using Clang [5]; (ii) simplify the code via constant propagation and dead code elimination after the code instrumentation; (iii) to apply further Clang optimizations (e.g., canonicalize natural loops and promote memory to register); (iii) add Map2Check library functions to check the analyzed LLVM bytecode; (iv) generate inputs for Map2Check instrumented functions by executing LibFuzzer and then KLEE with Crab-LLVM; and (v) generate the witness file by identifying each basic block executed in the control-flow graph of the LLVM IR.

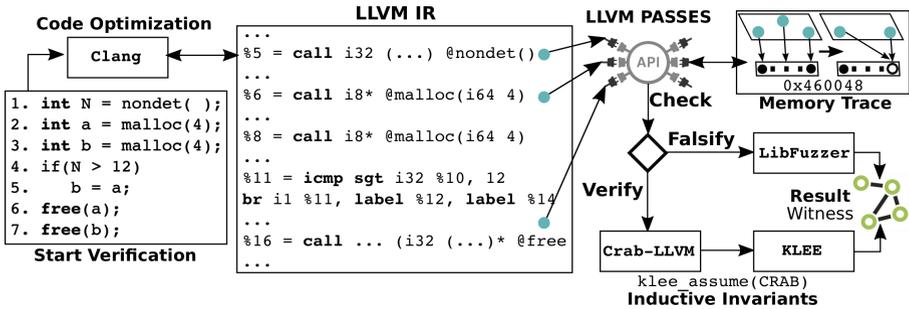


Fig. 1. Map2Check Verification Flow.

In order to explore the program states and to generate inputs for the Map2Check instrumented functions, the LibFuzzer implementation works by creating a custom entry point, which contains an array of bytes (of `uint8_t`). Thus, our implementation consists of generating concrete values from non-deterministic inputs that are our fuzzy targets. Additionally, we run multiple libFuzzer processes in parallel, where N fuzzing jobs should run to completion, i.e., until a bug is found or time/iteration limits are reached. Our fuzzing is coverage-guided (e.g., clang coverage), which tries to maximize the code coverage of a program. In our case, we adopted an `inline-8bit-counters` option from LLVM (SanitizerCoverage) for code coverage instrumentation built-in, where the compiler will insert inline counter that should be incremented on every edge.

The KLEE implementation works by creating a variable for the used data type, makes it symbolic, and then returns its value. As a result, KLEE produces concrete inputs for different program executions. We extend our KLEE implementation by adopting MetaSMT [6], which is an Embedded Domain Specific Language for SMT solvers. The API provided by MetaSMT is translated at compile-time, through template meta-programming, into the native APIs provided by the SMT solvers [9]. Therefore, the overhead introduced by MetaSMT is small.

In order to improve the KLEE core solver execution, the KLEE tool is ran adopting: counterexample caching solver, which can be used to avoid calling the underlying solver in certain situations; and MetaSMT, which is employed to construct expressions that will be cached for each constraint to facilitate expression reuse. Note that symbolic execution often requires concrete solutions for satisfiable queries, e.g., before calling an external function, all symbolic bytes need to be replaced by concrete values, simplify constraints, and reuse query results [9]. Therefore, the KLEE cache solver is an important optimization, mainly of the counterexample cache that is based on the observation that many constraint sets are in a subset/superset relation.

To check the unreachability of an error location, we reduced the number of states in the analyzed program to be explored, thereby supplying invariants to the back-end solvers. We adopted Crab-LLVM [11] to infer inductive invariants as constraints to the error location. Therefore, the invariants are automatically introduced into the program as assumptions (before verification), and then KLEE receives the code as input. Crab-LLVM is a static analyzer that employs an abstract interpretation engine over LLVM bitcode based on the Crab library, which uses abstract domains such as intervals, octagon, and polyhedra. Crab is built on the top of IKOS¹ (Inference Kernel for Open Static Analyzers) to support a collection of abstract domains and fixpoint iterators.

3 Software Architecture

Map2Check v7.3.1 is implemented as a source-to-source transformation tool in C/C++ using LLVM (v6.0). Map2Check uses Clang (v6.0) as a front-end to parse a C program and to generate the respective LLVM bitcode to be used in the code transformation to track pointers and variable assignments. It uses LibFuzzer [7] (v6.0) and KLEE [3] (v2.0, as a symbolic execution) to automatically produce inputs to execute different program paths. MetaSMT (v4.rc2) is the API of reasoning engines. For SV-COMP'20, we adopt Yices (v2.5.1) that is used by KLEE to check constraints over bit-vectors and arrays, which substantially improved our results. Crab-LLVM [11] is used on reachability mode to infer inductive invariants for LLVM bitcode.

4 Strengths and Weaknesses of the Approach

Map2Check analyzed intricate verification tasks. The tool achieved the 2nd place in the ReachSafety-Arrays subcategory; in the ReachSafety-BitVectors category, Map2Check achieved a score of 46, thereby presenting better results than Pinaka, UKojak, VeriFuzz, and DIVINE. In other subcategories, our tool generated correct-unconfirmed and incorrect true results. These results are, in part, explained due to the Map2Check bugs in the witness generation and limitation to handle Crab-LLVM invariants from the over-approximations. We are investigating how to extend our tool by combining the data from fuzzing with KLEE as program assumptions using template invariant.

In the MemSafety category, Map2Check achieved a score of -68. However, our tool achieved essential results in comparison with the state-of-art tools, e.g., in the

¹ <https://ti.arc.nasa.gov/opensource/ikos/>

MemSafety-heap subcategory achieved a score of 174, which outperforms UAutomizer, ESBMC, DIVINE, and CBMC. Most incorrect results are, in part, explained due to bugs in the pointer tracking from our memory model, which could be improved by a trace semantics with program optimizations as relations on sets of the trace. Sadly, in the NoOverflows category, the score was -89 . The incorrect results are, in part, explained due to bugs in the overflow analyzer. One way to improve this result is by combining the CPU flag postcondition test (LLVM supports several intrinsic functions, e.g., an add operation returns a structure with the result and overflow flag) with Sanitizers checking.

5 Tool Setup and Configuration

In order to run our `map2check-wrapper.py` script [10],² one must set the property file (`-p`) and the verification task; it provides as result: *TRUE + Witness*, *FALSE + Witness*, or *UNKNOWN*. For each error-path or correctness witness, a file (called `witness.graphml`) with the witness proof is generated in the Map2Check root-path folder. The dependencies, e.g., Clang and Yices tools, are included in the Map2Check distribution. The Benchexec tool info module is named `map2check.py` and Map2Check participates in SV-COMP'20 (as in the `map2check.xml` benchmark definition) in the following categories: ReachSafety-Arrays, ReachSafety-BitVectors, ReachSafety-ControlFlow, ReachSafety-Heap, ReachSafety-Loops, ReachSafety-Recursive, MemSafety, and NoOverflows.

6 Software Project

Map2Check v7.3.1³ is open source software distributed under the GPL license. We provide instructions for building Map2Check from the source in the file README (including the description of all dependencies). Map2Check is a joint project with the Federal University of Roraima and the Federal University of Amazonas in Brazil.

References

1. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51(3) (2018)
2. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: TACAS. pp. 174–177. Springer (2009)
3. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI. pp. 209–224. USENIX (2008)
4. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV. pp. 737–744. Springer (2014)
5. Fandrey, D.: Clang/LLVM Maturity Report. In: Computer Science Dept., University of Applied Sciences Karlsruhe (2010), See <http://www.iwi.hs-karlsruhe.de>.
6. Haedicke, F., Frehse, S., Fey, G., Große, D., Drechsler, R.: metaSMT: Focus on Your Application not on Solver Integration. In: Intl. Workshop on DIFTS. CEUR-WS.org (2011)

² <https://gitlab.com/sosy-lab/sv-comp/archives-2020/blob/master/2020/map2check.zip>

³ <https://github.com/hbgit/Map2Check>

7. LibFuzzer: A library for coverage-guided fuzz testing. <https://llvm.org/docs/LibFuzzer.html> (2019), [Online; accessed September-2019]
8. Menezes, R., Rocha, H., Cordeiro, L., Barreto, R.: Map2Check using LLVM and KLEE. In: TACAS. pp. 437–441. Springer (2018)
9. Palikareva, H., Cadar, C.: Multi-solver Support in Symbolic Execution. In: Intl. Workshop on SMT. p. 15. CEUR-WS.org (2014)
10. Rocha, H., Menezes, R., Cordeiro, L.C., Barreto, R.: Map2Check Tool: Using Symbolic Execution and Fuzzing. Zenodo. (Feb 2020), <https://doi.org/10.5281/zenodo.3678748>
11. SeaHorn: Crab-LLVM: Abstract Interpretation of LLVM bitcode. <https://github.com/seahorn/crab-llvm> (2019), [Online; accessed November-2019]

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

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





PredatorHP Revamped (Not Only) for Interval-Sized Memory Regions and Memory Reallocation (Competition Contribution) *



Petr Peringer, Veronika Šoková** (✉), and Tomáš Vojnar

Brno University of Technology, Faculty of Information Technology,
Centre of Excellence IT4Innovations, Czech Republic

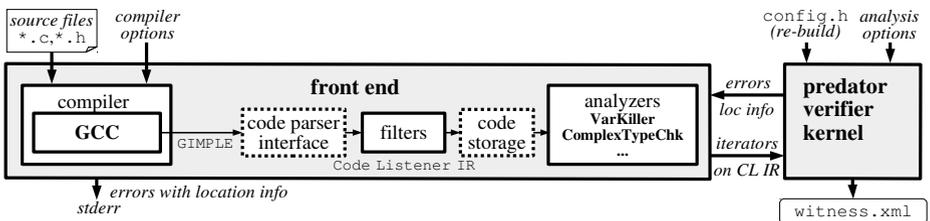
Abstract. This paper concentrates on improvements of the PredatorHP shape analyzer in the past two years, including, e.g., improved handling of interval-sized memory regions or new support of memory reallocation. The paper characterizes PredatorHP’s participation in SV-COMP 2020, pointing out its strengths and weakness and the way they were influenced by the latest changes in the tool.

1 Verification Approach and Software Architecture

We first briefly recap the main ideas behind PredatorHP and then discuss significant improvements that have been done in the tool in the past two years.

1.1 The Predator Shape Analyzer

Predator is implemented using C++ and the Boost libraries as a GCC plug-in on top of the Code Listener framework [2], which we recently upgraded to work with GCC 7.4.0. Moreover, as shown below, we extended Code Listener by adding a type analysis phase before the compiled code is passed to the shape analysis implemented in Predator. In case a memory safety property is to be checked and there are no complex types, such as structures, unions, arrays, strings, or pointers in the program under analysis (including possibly unreachable code), we directly assume the program to be memory safe.



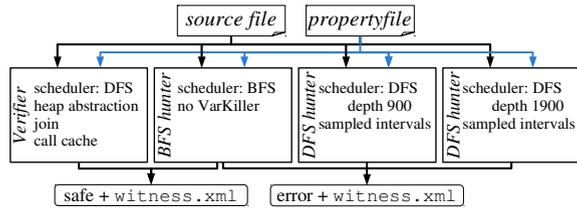
The main aim of Predator is *shape analysis* of sequential C programs that use low-level C pointer statements to implement various kinds of lists (singly- or doubly-linked, possibly circular, nested, and/or shared). Predator looks for various *memory-related errors* (invalid pointer dereferences, double free operations, memory leaks, etc.), and it

* This work was supported by the Czech Ministry of Education, Youth and Sports within the IT4Innovations Excellence in Science (NPUII) project No. LQ1602.

** Jury member, email: isokova@fit.vutbr.cz.

also checks validity of *assertions* present in the code. Predator uses *abstract interpretation* based on the domain of *symbolic memory graphs* (SMGs) [1]. Predator abstracts uninterrupted sequences of singly- or doubly-linked memory regions into appropriate kinds of list segments. Further, Predator abstracts numerical values (either values stored in memory regions, sizes of the regions, or offsets of pointers) using intervals with constant bounds. The constants used as the bounds have a pre-defined maximum/minimum value defined in the configuration of Predator (+32/-32 for SV-COMP’20). If the maximum/minimum value is exceeded, the bound is set to plus or minus infinity. Predator uses *summaries* to speed up analysis of programs structured into functions. Recursive programs are, however, analysed up to a given call depth only.

PredatorHP, i.e., the *Predator Hunting Party* [3,4], whose flow of control is shown on the right, is implemented as a Python script, and used to increase the efficiency and precision of the analysis. Namely,



PredatorHP runs the base *Predator verifier* in parallel with several *Predator hunters* that do not use the list-segment abstraction, do not join semantically different SMGs, nor use function summaries with matching of call parameters based on SMG entailment. While the Predator verifier can claim a program correct, it cannot report errors to avoid false alarms caused by abstraction. Predator hunters are classified as *breadth-first* (BFS) and *depth-first* (DFS). The DFS hunters have a limit on the search depth defined as a certain number of GCC’s GIMPLE instructions. The hunters can normally only report errors. The only exception is when the verified program has a finite state space that is fully explored by the BFS hunter in the given time limit.

In SV-COMP’20, based on empirical data, the BFS hunter does not use the Predator’s *VarKiller*, which removes dead variables from SMGs. This led to a significant speedup on 5 verification tasks (and some slowdown on 3 tasks). Further, the most shallow DFS 200 hunter, searching up to the depth of 200 instructions and used in PredatorHP up to SV-COMP’19, was removed as it was not bringing any advantage wrt the DFS 900 hunter, and a DFS 1900 hunter was added to handle more complex tasks (in particular, `memsafety-ext2/split_list_test05-1,ntdrivers/floppy.i.cil-3`). However, note that the DFS 900 hunter remains needed as otherwise 11 verification tasks would time out.

1.2 Recent Modifications of PredatorHP

One of the main improvements of the latest version of Predator is that its SMG-based analysis has been extended to support *memory reallocation* on the heap. If a reallocation operation is executed on an SMG, two new SMGs are produced. The first one models the case when a new object of the required size is created, data from the old object are copied into the new object, and the old object is freed. In the second case, the existing object is resized. If the size decreases, Predator checks that no memory leak happens due to some pointer field is removed or invalidated (in case it is partially removed).

Another improvement concerns working with *interval-sized memory regions*, which arise when allocating structures or arrays of parametric size. Despite even older versions

of Predator were able to create such regions, the way in which they could have been treated in the subsequent analysis of the program was very limited. In particular, it was impossible to dereference interval-sized regions, and hence Predator was very weak when analysing programs with structures or arrays of an in-advance-not-fixed size. This situation was first improved for SV-COMP'19 in the following pragmatic way.

Namely, whenever Predator hits a conditional statement that would previously yield an interval value with fixed bounds (such as the statement `if (n>=0 && n<10)` for so-far unconstrained `n`), it will split the further analysis into as many branches as the number of values in the interval is, each of them evaluating for a concrete value from the interval. After the split, no further interval-based allocations and dereferences, which the previous version of Predator used to fail on, happen. In order for the splitting not to cause a memory explosion, the latest version of Predator contains a parameter that controls the maximum size of split intervals, which was set to 300 in SV-COMP'20.

The above modification of Predator concerned dealing with memory regions whose size is given by an interval with finite bounds. In case one of the bounds is infinite, Predator has been extended to *sample* the interval and perform the further analysis with the sampled values. Currently, the sampling is done simply by taking some number of concrete values from the given interval starting/ending with the bound that is fixed (of course, for memory regions, unboundedness from above does only make sense). The number of considered samples is currently set to 3. Of course, this strategy cannot be used to soundly verify correctness of programs, and so it is used for detecting bugs only.

Despite the above mentioned treatment of intervals was primarily designed for dealing with interval-sized memory regions, it can help in other cases of dealing with integers too. Namely, it can help both when dealing with integer data as well as when dealing with interval-based pointer offsets.

Next, we have implemented checking whether all dynamically allocated memory has been deallocated when a function with the *noreturn attribute* (such as `abort` or `exit`) is called. The implementation simply searches the SMG representing the memory at the moment of a call of a noreturn function and checks that it does not contain any valid dynamically allocated object.

We have also added a support of the *clobber* instruction of GIMPLE, which terminates the life time of local variables of code blocks. Upon this instruction, Predator now marks the concerned memory region as deallocated, allowing it to detect *invalid dereferences of objects local to a block* from outside of the block. Further, we have added a support of the instructions *modulo* and *bitwise-or* and created models of the standard library functions for `strcmp` and `realloc`. This fixed several problems such as reporting false alarms when assigning fully-overlapping structures.

Finally, we improved the generation of witnesses. Apart from some bug fixes, we changed the trace generation for the reachability category. Namely, in this category, if some trace ends with an error other than calling `__VERIFIER_error`, the analysis recovers and continues to search for other traces.

2 Strengths and Weaknesses

The main strength of PredatorHP is that it treats code with various kinds of unbounded lists in a *sound* and *efficient* way. Predator hunters then allow it to quickly handle programs with a small finite state space (e.g., benchmarks from `list-simple`) and avoid

many false alarms that could otherwise happen. Interestingly, among the 328 correct tasks in *ReachSafety-Heap*, *MemSafety-Heap*, and *MemSafety-LinkedLists*, only 98 use unbounded data structures, out of which the Predator verifier (and, of course, no hunter) handles 56 %. Next, out of the 328 tasks, 83 do not use linked data structures nor arrays, and 147 use them but are finite-state. The Predator verifier and the BFS hunter handle 93 % of the 83 tasks that are so trivial that even the verifier does not use any abstraction. Out of the 147 tasks, 53 tasks are handled by both of them, while 2 tasks are handled solely by the verifier and 75 solely by the BFS hunter.

A weakness of Predator is that it specialises in dealing with lists, and so it handles structures such as trees, skip-lists, or arrays in a bounded way, i.e., for error detection, only. Another weakness of Predator has traditionally been its weak treatment of non-pointer data. We have tried to improve on the latter weakness by the described heuristics for dealing with intervals of integers with a specific aim to improve the way Predator handles memory regions of parametric size. The results of PredatorHP on SV-COMP'20 benchmarks with arrays show that the heuristics did help. Indeed, the interval sampling heuristic allowed us to correctly detect 10 errors in tasks from `array-memsafety`, `array-examples`, and `loops`. Moreover, the interval-splitting heuristic also helped on some benchmarks for dealing with interval-based sizes, offsets, and/or integer data. Namely, it removed 8 unknown results in *ReachSafety* and 4 such results in *MemSafety*.

The new type analysis looking for presence of complex types allowed Predator to skip its main analysis loop in 77 tasks in the *MemSafety* category, of which 13 tasks (from `termination-crafted`) contain recursion, which Predator could not handle, and 6 tasks (from `locks`) would otherwise timeout. Due to the new support of reallocation, Predator verifies all tasks containing a call of `realloc`. Due to the added support of `clobber` instructions, Predator detects invalid memory accesses in benchmarks accessing variables outside of the block in which they were created. All other new improvements described above did also help in some cases and allowed PredatorHP to win the 1st place in the *MemSafety* category and in the *ReachSafety-Heap* sub-category.

3 Contributors, Software Project, and the Tool Setup

The main author of Predator is Kamil Dudka. Besides him and the PredatorHP team, Petr Müller, Michal Kotoun, and numerous other people listed in the `docs/THANKS` file in the distribution of Predator have contributed to the distribution of Predator.

Predator is an open source software project distributed under GNU GPLv3. The source code used in SV-COMP'20 is available too¹. The `README-SVCOMP-2020` file shipped with it describes how to build the tool. The script `predatorHP.py` serves to run the tool, taking a verification task file as a single positional argument. Paths to both the property file and the desired witness file are accepted via long options, i.e., 64-bit compiler options. The verification outcome is printed to the standard output. To run PredatorHP in the BenchExec environment, the `predatorhp.py` wrapper and the `predatorhp.xml` benchmark definition can be used. In SV-COMP'20, PredatorHP participated in the *MemSafety* category and in the *ReachSafety-Heap* sub-category.

¹ <http://www.fit.vutbr.cz/research/groups/verifit/tools/predator-hp>

References

1. Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 215–237. Springer, Heidelberg (2013)
2. Dudka, K., Peringer, P., Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2011, Part I. LNCS, vol. 6927, pp. 527–534. Springer, Heidelberg (2012)
3. Muller, P., Peringer, P., Vojnar, T.: Predator Hunting Party (Competition Contribution). In: Baier, C., Tinelli, C. (eds) TACAS 2015, LNCS, vol. 9035, pp. 443–446. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_40
4. Peringer, P., Šoková, V., Vojnar, T.: PredatorHP (Version 3.141). Zenodo (2020). <http://doi.org/10.5281/zenodo.3678356>

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

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





Symbiotic 7: Integration of Predator and More^{*}

(Competition Contribution)

Marek Chalupa^{1**}, Tomáš Jašek¹, Lukáš Tomovič¹,
Martin Hruška², Veronika Šoková², Paulína Ayaziová¹,
Jan Strejček¹, and Tomáš Vojnar²



¹ Masaryk University, Brno, Czech Republic

² Brno University of Technology, FIT,
IT4Innovations Centre of Excellence, CZ, Brno, Czech Republic

Abstract. SYMBIOTIC 7 brings improvements in all parts of the tool. In particular, we integrated the advanced shape analysis implemented in Predator to our instrumentation process for memory safety checking. Further, we extended our slicer to correctly handle non-terminating programs. This new slicing is applied in termination analysis, where we also added instrumentation for detection of simple cycles in the program state space. The witness generation process changed as well.

1 Verification Approach

SYMBIOTIC 7 follows the same basic schema as all previous versions [4,5]: the program to be verified is first instrumented (if needed), then reduced by static program slicing, and finally symbolically executed using KLEE [2]. We describe the main modifications since SYMBIOTIC 5 (participating in SV-COMP 2018) as modifications in SYMBIOTIC 6 (competing in 2019) have not been published.

Memory safety checking improvements SYMBIOTIC uses a static pointer analysis to detect instructions that can potentially violate memory safety. To check these instructions, SYMBIOTIC 5 [5,3] instrumented the program with code that keeps records about allocated memory and uses the records to assert the validity of potentially misbehaving instructions. Then we sliced the program with respect to these assertions and called KLEE to check assertion validity.

Since SYMBIOTIC 6, we slice the program directly with respect to the potentially misbehaving instructions without inserting any additional code. Then we call KLEE to check memory safety of the sliced program.

SYMBIOTIC 7 newly integrates PREDATOR [6], a static analyzer specialized on memory safety. We first run PREDATOR in its over-approximating mode and

^{*} M. Chalupa, T. Jašek, P. Ayaziová, and J. Strejček have been supported by the Czech Science Foundation grant GA18-02177S. M. Hruška, V. Šoková, and T. Vojnar have been supported by the IT4Innovations Excellence in Science project (LQ1602) and the FIT BUT internal project FIT-S-20-6427.

^{**} Jury member and corresponding author: chalupa@fi.muni.cz.

in a configuration that analyses all branches in the given program and tries to recover from found errors. If PREDATOR says that the program is safe, we simply answer *true*. Otherwise, we take bug reports from PREDATOR and combine them with results of our static pointer analysis to get a more precise (i.e., smaller) set of potentially misbehaving instructions. Then we proceed like SYMBIOTIC 6.

SYMBIOTIC 7 is also the first version that can distinguish between *valid-memcleanup* and *valid-memtrack* properties. To do this, our clone of KLEE now reconstructs the shape of memory at the program exit if unfreed memory is found: KLEE starts with local and global variables and resolves pointers in these (if any). Then it resolves pointers in the pointed memory, etc. This way we can find out if the unfreed memory is reachable via a chain of dereferences or not.

Termination analysis SYMBIOTIC 6 introduced a simple support for termination property: a call to `_VERIFIER_error` is inserted before trivial infinite loops, e.g., `while (true);` loops. If the symbolic execution detects that such a call is reachable, SYMBIOTIC answers *false* as the program can reach an infinite loop. If all paths of the program are explored by symbolic execution without reaching any of these calls, all program executions are clearly terminating and we answer *true* (an infinite program path cannot be fully explored by symbolic execution). Note that program slicing was disabled for non-termination checking in SYMBIOTIC 6 as the slicer could remove infinite loops in some specific cases.

SYMBIOTIC 7 brings two improvements. First, since we extended our slicer to correctly handle non-terminating programs [7], we now apply slicing with slicing criteria set to all exit points (including the instrumented error calls) of the program. Second, we instrument the program with checks for simple cycles in the state space. The instrumentation detects non-nested loops with a single entry for which it can conservatively determine a set $\{V_1, \dots, V_k\}$ that includes all variables potentially modified by the loop. At the beginning of the loop body, we insert assignments that store the value of each variable V_i into a new variable V'_i . At the end of the loop body, we insert the assertion `assert($V_1 \neq V'_1 \vee \dots \vee V_k \neq V'_k$)` to check a change in the vector of these variables. If this assertion is violated, the program has a non-terminating execution.

Error path replay Although the slicer in SYMBIOTIC now provides algorithms that preserve non-termination properties of programs, outside the *Termination* category we still use the original *non-termination insensitive* slicing as it may remove more instructions. The price is, however, that SYMBIOTIC may report false alarms: an unreachable error location situated below an infinite loop may become reachable when the loop is sliced out. To fix this issue, we try to reproduce each error found by symbolic execution in the original (unsliced) program. If the error is reproduced, we report it as a real error. Otherwise, we say *unknown*.

Improved witness generation SYMBIOTIC 5 and 6 generated violation witnesses that describe only the initialization of non-deterministic variables at the

beginning of the `main` function. SYMBIOTIC 7, on the other hand, generates violation witnesses that contain a complete test vector, i.e., the whole sequence of values returned from `__VERIFIER_nondet_*` functions during the error path replay. To get and correctly identify all these values, we have modified our fork of KLEE to support interpretation of `__VERIFIER_nondet_*` functions (and other undefined functions in general) internally. Currently, more than 99% of our violation witnesses (outside the *Termination* category) are confirmed. SYMBIOTIC 7 still generates trivial correctness witnesses if no error is found.

Other improvements Other improvements in SYMBIOTIC 7 used in SV-COMP 2020 include a faster data dependence analysis (a part of slicing) and better handling of `assume` statements in the slicer. SYMBIOTIC is now also able to continue in verification if the instrumentation or slicer crashes or exceeds the time limit. In such a case, KLEE is run on the original program which has been only optimized by standard LLVM optimizations. For SV-COMP 2020, we set the time limit of 400s on instrumentation and the time limit of 300s on slicing.

2 Software Architecture

SYMBIOTIC 7 is built on top of LLVM 8.0.1 [8]. The tool consists of a set of modules written in C++ that process LLVM bitcode, and Python scripts that chain these modules according to given configuration.

For use in SYMBIOTIC, we have made several bugfixes in PREDATOR’s LLVM backend and ported it to LLVM 8.0.1. Further, we have introduced distinguishing between safe and possibly erroneous program instructions.

SYMBIOTIC uses its own fork of KLEE that contains several modifications compared to the mainstream KLEE. In particular, the fork has been extended to handle symbolic-sized memory allocations, to process marks delimiting the lifetime of scoped variables, to check for memory leaks, and to generate violation witnesses in the SV-COMP format.

3 Strengths and Weaknesses

In SV-COMP 2020 [1], SYMBIOTIC 7 won the *SoftwareSystems* category and scored second in the *MemSafety* category and the *FalsificationOverall* meta category. Overall, SYMBIOTIC ended up on the fourth place.

The main reason for winning *SoftwareSystems* is having only a few incorrect answers. Indeed, SYMBIOTIC did not win in the number of correct answers in any of the *SoftwareSystems* subcategories. However, we had only 4 incorrect answers and all of them in the subcategory *DeviceDriversLinux64*. This subcategory is huge and these incorrect answers have only a small impact on the weighted score.

In *MemSafety*, we took the second place after PREDATORHP which executes several instances of the PREDATOR tool with different configurations in parallel. SYMBIOTIC calls just one of these instances as mentioned above. Additionally,

PREDATORHP uses GCC, while we use PREDATOR running on LLVM, which is not as mature as the former. Also, we had a number of new *unknown* answers because KLEE does not support pointer comparisons, which we incorrectly did not detect in the previous versions of SYMBIOTIC.

In general, SYMBIOTIC's results stems from the good performance of KLEE supported by efficient static analysis and slicing: the official results show that SYMBIOTIC can decide many benchmarks very quickly.

The main weakness of our tool is the inherent complexity of symbolic execution and the limited possibility of analysing potentially unbounded loops or infinite paths with this technique. Indeed, as symbolic execution actually follows all paths in the program, it does not terminate if the program contains an unbounded loop or an infinite path (unless an error is found). Even when the number of paths is finite and all the paths are finite, symbolic execution usually runs out of resources if the number of paths is large. Although this problem is slightly alleviated by program slicing, our tool still does not scale well on complex programs.

4 Tool Setup and Configuration

- *Download:* From the competition archives or via <http://doi.org/10.5281/zenodo.3678328>.
- *Installation:* Unpack the archive.
- *Participation Statement:* SYMBIOTIC 7 participates in all categories.
- *Execution:* Run `bin/symbiotic --sv-comp OPTS <source>`, where available OPTS include:
 - `--prp=file`, which sets the property specification file to use,
 - `--witness=file`, which sets the output file for the witness,
 - `--32`, which sets the 32-bit environment,
 - `--help`, which shows the full list of possible options.

5 Software Project and Contributors

SYMBIOTIC 6 and 7 have been developed by M. Chalupa, T. Jašek, M. Vitovská, M. Šimáček, L. Tomovič, and P. Ayaziová under the supervision of J. Strejček. Predator has been adjusted for the described integration by M. Hruška and V. Šoková under the supervision of T. Vojnar. SYMBIOTIC and its components are available under the MIT license. The project is hosted by the Faculty of Informatics, Masaryk University. KLEE, LLVM, and PREDATOR are also available under open-source licenses. Source codes of the project and references to all its components can be found at:

<https://github.com/staticafi/symbiotic>

References

1. D. Beyer. Advances in automatic software verification: SV-COMP 2020. In *Proc. TACAS (2)*, LNCS 12079. Springer, 2020.
2. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008.
3. M. Chalupa, J. Strejček, and M. Vitovská. Joint forces for memory safety checking. In M. Gallardo and P. Merino, editors, *SPIN*, volume 10869 of *LNCS*, pages 115–132. Springer, 2018. https://doi.org/10.1007/978-3-319-94111-0_7.
4. M. Chalupa, M. Vitovská, M. Jonáš, J. Slaby, and J. Strejček. Symbiotic 4: Beyond reachability (competition contribution). In A. Legay and T. Margaria, editors, *TACAS*, volume 10206 of *LNCS*, pages 385–389. Springer, 2017. https://doi.org/10.1007/978-3-662-54580-5_28.
5. M. Chalupa, M. Vitovská, and J. Strejček. Symbiotic 5: Boosted instrumentation (competition contribution). In D. Beyer and M. Huisman, editors, *TACAS*, volume 10806 of *LNCS*, pages 442–446. Springer, 2018. https://doi.org/10.1007/978-3-319-89963-3_29.
6. K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 372–378. Springer, 2011. https://doi.org/10.1007/978-3-642-36742-7_49.
7. L. Tomovič. Slicing of parallel programs. Master’s thesis, Masaryk University, 2019. <https://is.muni.cz/th/o1s3u/>.
8. LLVM. <http://llvm.org/>.

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

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





Ultimate Taipan with Symbolic Interpretation and Fluid Abstractions (Competition Contribution)

Daniel Dietsch^(✉) , Matthias Heizmann^(✉) ,
Alexander Nutz, Claus Schätzle, and
Frank Schüssele

University of Freiburg, Freiburg im Breisgau, Germany
{dietsch, heizmann}@cs.uni-freiburg.de



Abstract. ULTIMATE TAIPAN is a software model checker that combines trace abstraction with abstract interpretation on path programs. In this year's version, we replaced our abstract interpretation engine and now use a combination of multiple abstraction functions, fixpoint computation, algebraic program analysis, and SMT solving. Our new approach will allow us to integrate new techniques more easily.

1 Verification Approach

ULTIMATE TAIPAN is a software model checker which combines trace abstraction [8] and abstract interpretation [5]. The algorithm of TAIPAN follows the trace abstraction verification scheme for reachability where it constructs an abstraction of the program as a nested word automaton (NWA). This NWA has initially the same graph structure as the program's interprocedural control flow graph (ICFG), its states are program locations, its transitions are labeled with program locations, and states corresponding to error locations are accepting. Hence, the automaton recognizes a language where the symbols are statements and the words are sequences of statements (which we call traces) that lead to an error location. If the language of the abstraction automaton is empty, no error location can be reached and the program is safe. If there is a trace in the language, the algorithm needs to determine if it is a *feasible* trace, i.e., a trace that corresponds to an actual program execution, or not. Feasible traces constitute an actual counterexample and if one is found the algorithm terminates. If an infeasible trace is found, TAIPAN's algorithm differs from trace abstraction and does not only analyze the actual trace, but rather constructs a path program¹ from this trace. It then tries to synthesize inductive invariants for the whole path program [7]. From these invariants, a new automaton is constructed which language only recognizes infeasible traces. The new abstraction is then constructed as the difference of the automaton that only recognizes infeasible traces and the old abstraction automaton. If the error location's invariant of the path program is not *false*, the computed invariants are too weak to prove infeasibility, and TAIPAN falls back to using interpolating SMT solvers to compute new invariants that are strong enough to discharge the trace.

Daniel Dietsch — Jury Member

¹ A path program is a projection of the program to the trace.

© The Author(s) 2020

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12079, pp. 418–422, 2020.

https://doi.org/10.1007/978-3-030-45237-7_32

TAIPAN’s old algorithm used abstract interpretation to analyze path programs. In this year’s iteration, we use a new approach, which is motivated by two drawbacks of our old algorithm. Firstly, extending an abstract interpretation engine with new abstract domains is labor-intensive and error-prone. Each abstract domain has an abstract post operator describing the effect program statements have on abstract states. For each abstract domain and each type of program statement the abstract post operator has to be defined and implemented, and re-use between domains is complicated. Furthermore, each abstract domain needs their own representation of an abstract state, s.t. exchanging information between multiple domains requires explicit conversions. Secondly, Abstract interpretation always abstracts. Because each abstract domain has its own abstract state representation, it is usually not possible to implement a precise post operator. Hence, every application of post is an abstraction, which leads to unnecessary loss of precision.

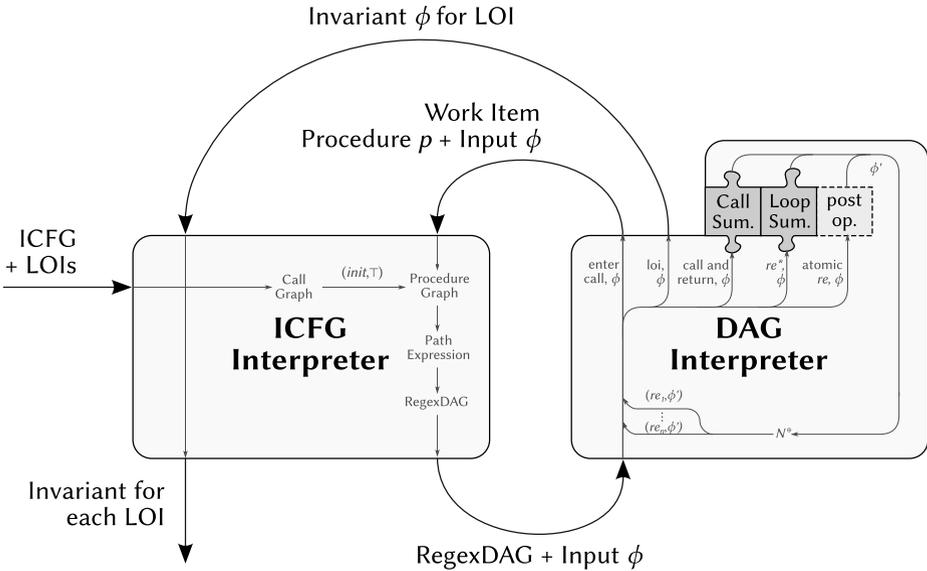


Fig. 1: Overview of the symbolic interpretation engine.

Our new approach is inspired by *Algebraic Program Analysis* [9, 4] and the renewed interest in this technique (e.g. [6]), and *Logical Interpretation* [10]. We use the modularity of algebraic program analysis to combine different techniques in an unifying framework and the idea of a shared representation of abstract program states as SMT formulas over which abstraction operators can compute fixpoints from logical interpretation.

An overview of our approach is depicted in Figure 1. The approach consists of two major components, the ICFG interpreter and the DAG interpreter.

The ICFG interpreter component generates for a (partial) interprocedural control flow graph (ICFG) and a subset of its program locations (locations of interest, LOI) a set of path expressions represented as RegexDAGs. A RegexDAG is a di-

rected acyclic graph with vertices that are labeled with regular expressions over the program’s statements without calls and returns but with summary and enter statements. Each RegexDAG has exactly one sink node that represents a location of interest. We use summary statements when we call to and return from a procedure on a path to a LOI, and enter statements when we do not return until we reach the LOI.

The DAG interpreter component then analyses a RegexDAG in topological order by applying different operators (Call Sum., Loop Sum., post op.) to the different vertex labels. All operators take a program state expressed as SMT formula ϕ and a regular expression over program statements (i.e., a vertex label) and produce a new (possibly abstracted) program state that captures all the effects. If a vertex has multiple incoming edges, the different input states are simply joined with a logical disjunction (\vee). Some of these operators depend again on the ICFG interpreter to compute their result. The most basic operator is the post operator (post op.), which computes strongest post for star-free regular expressions and optionally applies an abstraction function to the result. The choice of abstraction function and if to apply them is governed by different heuristics that can be changed. We call these heuristics *fluids*. The other operators are the call summarization (Call Sum.) and loop summarization (Loop Sum.) operators. The call summarization operator computes a summary for a procedure call, either with or without considering the context. The loop summarization operator computes a summary for the Kleene-star operator of regular expressions. Our current implementation does this by computing a fixpoint and resolving nested loops by recursively inserting summaries. The different operators (post, call summarization, loop summarization) are completely modular and can be considered black-boxes for the interplay between the two main components. When the DAG interpreter reaches the sink vertex of the RegexDAG, it returns the disjunction of this sink’s input program states as invariant for this LOI.

2 Strengths and Weaknesses

Our new approach is easy to extend with new abstraction functions, fluids, and loop acceleration techniques. Compared to the previous approach we also gain much more precision by, e.g., having a reduced product between different kinds of abstraction without writing a transformation function – we can just use the logical disjunction. Using SMT formulas as representation of program states also allows us to reuse many of ULTIMATE’s existing tools that deal with SMT, in particular simplification, quantifier elimination, rewriting, and debugging.

Nevertheless, our current implementation is not as effective as the old one, because we did not finish porting the various abstract domains. We currently only support a basic interval abstraction and an explicit value abstraction, which severely limits the efficiency of our approach. We are also missing more intricate loop acceleration implementations, optimized fluid configurations, and our implementation does not yet support recursion.

3 Architecture, Setup, Configuration, and Project

ULTIMATE TAIPAN is a part of the open-source program analysis framework ULTIMATE^{2,3}, written in Java and licensed under LGPLv3⁴. We used TAIPAN version 0.1.25-f470102c in our competition submission, which is available as a .zip archive from multiple sources^{5,6,7}. Our submission requires Java 1.8 and Python 3.x. The submission contains an executable version of TAIPAN for Linux platforms, the binaries of the required SMT solvers Z3⁸, CVC4⁹, and MATHSAT¹⁰, as well as a Python script, `Ultimate.py`, which maps the SV-COMP interface to ULTIMATE's command line interface. TAIPAN is invoked with

```
./Ultimate.py --spec prop.prp --file input.c --architecture
32bit|64bit --full-output
```

where `prop.prp` is the SV-COMP property file, `input.c` is the input C file, `32bit` or `64bit` is the architecture, and `--full-output` enables verbose output to `stdout`. The output of TAIPAN is written to the file `Ultimate.log`. A violation [3] or correctness [2] witness may be written to the file `witness.graphml`. The benchmarking tool BENCHEXEC [1] supports TAIPAN through the tool-info module `ultimatetaipan.py`¹¹. TAIPAN participates in all categories, as declared in its SV-COMP benchmark definition file `utaipan.xml`¹².

References

- [1] D. Beyer. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *TACAS 2016*, pages 887–904, 2016.
- [2] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness Witnesses: Exchanging Verification Results between Verifiers. In *FSE 2016*, pages 326–337, 2016.
- [3] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness Validation and Stepwise Testification across Software Verifiers. In *ESEC/FSE 2015*, pages 721–733, 2015.
- [4] J. A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, 1964.
- [5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL 1977*, pages 238–252, 1977.
- [6] J. Cyphert, J. Breck, Z. Kincaid, and T. W. Reps. Refinement of Path Expressions for Static Analysis. *PACMPL*, 3(POPL):45:1–45:29, 2019.
- [7] M. Greitschus, D. Dietsch, and A. Podelski. Loop Invariants from Counterexamples. In *SAS 2017*, pages 128–147, 2017.
- [8] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of Trace Abstraction. In *SAS 2009*, pages 69–85, 2009.
- [9] R. E. Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28(3):577–593, 1981.

² <https://ultimate.informatik.uni-freiburg.de>

³ <https://github.com/ultimate-pa/ultimate>

⁴ <https://www.gnu.org/licenses/lgpl-3.0.en.html>

⁵ <https://gitlab.com/sosy-lab/sv-comp/archives-2020/blob/master/2020/utaipan.zip>

⁶ <https://github.com/ultimate-pa/ultimate/releases/download/v0.1.25/UltimateTaipan-linux.zip>

⁷ <https://doi.org/10.5281/zenodo.3678625>

⁸ <https://github.com/Z3Prover/z3>

⁹ <https://cvc4.cs.nyu.edu/>

¹⁰ <http://mathsat.fbk.eu/>

¹¹ <https://github.com/sosy-lab/benchexec/blob/master/benchexec/tools/ultimatetaipan.py>

¹² <https://github.com/sosy-lab/sv-comp/blob/master/benchmark-defs/utaipan.xml>

- [10] A. Tiwari and S. Gulwani. Logical Interpretation: Static Program Analysis Using Theorem Proving. In *CADE*, volume 4603 of *LNCS*, pages 147–166. Springer, 2007.

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

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



Author Index

- Abate, Alessandro I-97
Afzal, Mohammad II-383
Ahmed, Daniele I-97
Akshay, S. I-387
Albert, Elvira II-118
Almaawi, Alyas I-115
Almeida, Bernardo II-39
An, Jie I-444
Angluin, Dana II-325
Ayaziová, Paulína II-413
- Babu M, Charles II-383
Baier, Christel I-324
Barreto, Raimundo II-403
Barrett, Clark I-367
Bartocci, Ezio I-492
Becker, Benedikt II-235
Bendík, Jaroslav I-135
Benerecetti, Massimo II-289
Berger, Philipp I-40
Beyer, Dirk I-3, II-126, II-347
Biagi, Marco I-463
Bian, Jinting II-217
Bockenek, Joshua A. II-98
Boender, Jaap II-271
Bornat, Richard II-271
Bozga, Marius I-228
Budde, Carlos E. I-463, I-483
- Castro, David II-278
Celik, Ahmet II-137
Černá, Ivana I-135
Chakraborty, Supratik I-22, II-383
Chalupa, Marek II-413
Chauhan, Avriti II-383
Chen, Mingshuai I-444
Chimdyalwar, Bharti II-383
Cimatti, Alessandro I-155
Cordeiro, Lucas C. II-403
Correas, Jesús II-118
Cubuktepe, Murat I-287
- D'Argenio, Pedro R. I-463
Dangl, Matthias I-3
Darke, Priyanka II-383
de Boer, Frank S. II-217
de Gouw, Stijn II-217
Delgrange, Florent I-346
Dell'Erba, Daniele II-289
Deng, Yuxin II-21
Dietsch, Daniel II-418
Dixon, Alex I-405
Du, Wenjie II-21
Dubut, Jérémy I-191
- Esparza, Javier I-228
- Fan, Chuchu I-173
Fedyukovich, Grigory II-195
Ferreira, Francisco II-278
Fisman, Dana II-325
Frenkel, Hadar I-211
Frohn, Florian I-58
Funke, Florian I-324
Furbach, Florian II-378
- Gastin, Paul I-387
Geatti, Luca I-155
Geldenhuis, Jaco II-373
Giacobbe, Mirco II-79
Gligoric, Milos II-137
Goel, Aman I-413
Gordillo, Pablo II-118
Griggio, Alberto I-155
Groote, Jan Friso II-3
Grumberg, Orna I-211
Gupta, Aarti II-195
Gupta, Ashutosh I-22, II-383
- Hahn, Ernst Moritz I-306
Hamers, Ruben I-266
Hasuo, Ichiro I-191
Heizmann, Matthias II-418

- Heljanko, Keijo II-378
 Henzinger, Thomas A. II-79
 Hiep, Hans-Dieter A. II-217
 Howar, Falk II-398
 Hruška, Martin II-413
 Huisman, Marieke I-247
 Hussein, Soha II-393

 Iosif, Radu I-228

 Jansen, David N. II-3
 Jansen, Nils I-287
 Jantsch, Simon I-324
 Jašek, Tomáš II-413
 Jeannerod, Nicolas II-235
 Jongmans, Sung-Shik I-266
 Joosten, Sebastiaan J. C. I-247
 Junges, Sebastian I-287

 Kammüller, Florian II-271
 Kápl, Roman II-254
 Katoen, Joost-Pieter I-40, I-287, I-346
 Katsumata, Shin-ya I-191
 Keiren, Jeroen J. A. II-3
 Khurshid, Sarfraz I-115
 Kimberly, Greg I-155
 King, Andy I-79
 Kobayashi, Naoki II-195
 Kolčák, Juraj I-191
 Kovács, Laura I-492
 Krishna, S I-387
 Kumar, Shrawan II-383

 Lang, Frédéric II-57
 Lazić, Ranko I-405
 Lechner, Mathias II-79
 Lochmann, Alexander II-178

 Maathuis, Olaf II-217
 Madhusudan, P. II-158
 Malík, Viktor II-368
 Mann, Makai I-367
 Manolios, Panagiotis II-388
 Marché, Claude II-235
 Mateescu, Radu II-57
 Mathur, Umang II-158
 Mazzanti, Franco II-57
 McCamant, Stephen II-393
 Meel, Kuldeep S. I-115

 Menezes, Rafael II-403
 Meyer, Roland II-378
 Middeldorp, Aart II-178
 Mitra, Sayan I-173
 Mogavero, Fabio II-289
 Mokhlesi, Navid I-173
 Monti, Raúl E. I-463
 Mordido, Andreia II-39
 Mues, Malte II-398
 Mutius, Joshua von I-425

 Nagarajan, Rajagopal II-271
 Neele, Thomas II-307
 Nutz, Alexander II-418

 Okudono, Takamasa I-79
 Oortwijn, Wytse I-247

 Palmskog, Karl II-137
 Parížek, Pavel II-254
 Pasareanu, Corina I-211
 Perez, Mateo I-306
 Peringer, Petr II-408
 Peruffo, Andrea I-97
 Poly, Guillaume II-271
 Ponce-de-León, Hernán II-378

 Qin, Xudong II-21
 Quatmann, Tim I-346
 Quiring, Benjamin II-388

 Randour, Mickael I-346
 Ravindran, Binoy II-98
 Régis-Gianas, Yann II-235
 Rocha, Herbert II-403
 Román-Diez, Guillermo II-118
 Roychowdhury, Sparsa I-387
 Rubio, Albert II-118

 Sakallah, Karem I-413
 Schätzle, Claus II-418
 Schewe, Sven I-306
 Schrammel, Peter II-368
 Schüssele, Frank II-418
 Sharma, Vaibhav II-393
 Sheinvald, Sarai I-211
 Shoval, Yaara II-325
 Sibai, Hussein I-173
 Sifakis, Joseph I-228

- Sighireanu, Mihaela II-235
Šoková, Veronika II-408, II-413
Somenzi, Fabio I-306
Sprunger, David I-191
Stankovič, Miroslav I-492
Stoelinga, Mariëlle I-463
Strejček, Jan II-413
Švejda, Jan I-40
- Tomovič, Lukáš II-413
Tonetta, Stefano I-155
Topcu, Ufuk I-287
Treinen, Ralf II-235
Trivedi, Ashutosh I-306
- Unadkat, Divyesh I-22, II-383
Usman, Muhammad I-115
- van de Pol, Jaco I-247
van Eekelen, Marko II-217
Vasconcelos, Vasco T. II-39
Venkatesh, R II-383
- Verbeek, Freek II-98
Visser, Willem II-373, II-393
Viswanathan, Mahesh II-158
Vojnar, Tomáš II-368, II-408, II-413
- Wang, Kaiyuan I-115
Wang, Wenxi I-115
Welzel, Christoph I-228
Wendler, Philipp II-126
Wesselink, Wieger II-307
Whalen, Michael W. II-393
Wijs, Anton II-3
Willemse, Tim A. C. II-307
Wimmer, Simon I-425
Wojtczak, Dominik I-306
- Yamada, Akihisa I-191
Yoshida, Nobuko II-278
- Zhan, Bohua I-444
Zhan, Najun I-444
Zhang, Miaomiao I-444